

# TI – DSP 多核技术 及实时软件开发

潘 晔 廖昌俊 编著

電子工業出版社·

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

本书从 DSP 软件开发的各个角度阐述了 TI 公司提供的 DSP 软件技术和开发工具，为 DSP 软件开发人员理清思路，以简化和加快 DSP 系统的软件开发。第 1 章首先从宏观上讨论了 DSP 嵌入式系统软件开发应注意的要素，然后简介了 TI 公司的 eXpressDSP 实时软件组件和开发工具。第 2~5 章分别从 DSP 可重用实时软件技术、多核嵌入式软件开发、优化的 DSP 库，以及 DSP 软件开发工具等几方面进行了详细介绍。

本书所涉及的材料，是截止到 2014 年的最新资料。结合编者的项目开发经验，增加了实现的例子，有利于读者理解和应用。本书可以作为电子信息类专业研究生和高年级本科生的参考书，对从事 DSP 技术研究和开发的科研人员和工程技术人员也很有参考价值。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

## 图书在版编目 (CIP) 数据

TI - DSP 多核技术及实时软件开发/潘晔，廖昌俊编著. —北京：电子工业出版社，2016. 1  
(DSP 应用丛书)

ISBN 978-7-121-27635-4

I. ①T… II. ①潘… ②廖… III. ①数字信号处理 - 程序设计 IV. ①TN911. 72

中国版本图书馆 CIP 数据核字 (2015) 第 281631 号

责任编辑：曲 昕 特约编辑：张传福

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1 092 1/16 印张：20.25 字数：531 千字

版 次：2016 年 1 月第 1 版

印 次：2016 年 1 月第 1 次印刷

定 价：49.80 元

凡所购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

# 序

自 20 世纪 70 年代末 80 年代初，DSP 处理器诞生以来，其发展与推广应用神速，在短短的 30 多年时间内，应用的领域和深度，令人叹为观止。

随着科学技术的进步，尤其是微电子和软件科学与技术的发展，微处理器的种类、型号与性能的发展，只能用“眼花缭乱”来形容。各类微处理器之间的技术融合、功能交叠，一方面使人们有了更多的选择余地，但也使得制订系统方案时的选择出现了方方面面的困难，主要是权衡利弊、取优舍劣、软硬件性能与开发难易程度的选择，性能价格比的考虑，以及发展前景的预测等。

微处理器（包括 DSP 处理器）硬件性能的极大改善，为软件的开发提供了很大的余地和空间；软件技术的进步也为微处理器软件的开发提供了极大的方便。特别值得一提的是编译器的优化，极大地提高了高级语言编译的效率，使其结果的优化程度，可以和直接用汇编语言程序的编译结果相媲美。也就是说，编程人员完全可以从烦琐的汇编语言编程工作中解放出来，使用自己熟悉的高级语言来编程，工作难度的降低和效率的提高，不言而喻。

为了推广自己公司的产品，各微处理器厂商还不断地推出、更新和优化自己系列处理器的开发工具和算法库，使应用系统的软件开发人员得以方便和高效地开展工作。

仍然存在不方便的是，不同厂商微处理器的硬件和软件系统，以及开发环境和工具，各不相同。因此，应用系统的开发人员，在系统开发之初，必须谨慎地选择所要使用的微处理器；不但是这一代产品，还得考虑后续的产品，因为改变所使用的微处理器，成本极为高昂，除了硬件、软件和开发环境的成本，还有开发所投入的人力成本，以及推迟新产品上市的机会成本，等等。

本书是针对德州仪器（TI）公司的 DSP 软件开发而编写的。如上所述，各家公司的硬件系统、软件系统和开发环境，各不相同。即便如此，基本的思路和方法还是一致的。有经验的开发人员都有这样的体会，真正熟悉了一家公司的东西，即便改用其他公司的产品上手也很容易，就是这个道理，即所谓“举一反三”。

参与本书编写的几位作者，都是电子科技大学的教师，是在 DSP 技术领域工作多年的资深教师和研究人员。他们有很好的“数字信号处理”的理论功底，熟悉 DSP 的硬件系统、软件系统和开发环境与工具，完成过多项包含 DSP 处理器的复杂系统的研制，因此，他们拥有完善的相关知识，积累了丰富的工作经验。

本书以实用为目的，基于作者的 DSP 工程开发经验，从 TI 公司纷繁复杂的文档中整理出有利于工程人员开发 DSP 系统的体系，为 DSP 软件开发人员理清思路。我相信，认真阅读和学习本书的读者，一定可以从中获得丰富的知识和体会，并在自己的学习和开发工作中，得益良多。

彭启琮

2015 年 11 月于

电子科技大学





# 前 言

随着数字信号处理（DSP）技术的发展，其应用无处不在。各种丰富多彩的多媒体智能终端带给人们方便快捷的应用体验，人们可以随时访问网络、处理音频和视频、规划交通路线等。除了上述消费类电子设备，工业控制、安防系统、通信系统、医疗设备、航天航空、军事装备等各方面都离不开 DSP。因此，DSP 软硬件开发以及系统集成等成为人们关注的问题。在通常情况下，开发一个 DSP 嵌入式系统，80% 的努力及 80% 的复杂度均取决于软件；如何提高 DSP 软件的开发速度、降低开发难度和成本至关重要。

目前，DSP 芯片的功能越来越复杂，多核片上系统（SOC）普遍应用，外设种类越来越多，大量新技术标准、新算法、新应用层出不穷。开发人员要花很长的时间来熟悉各种标准，而这些标准又在不停地改变。已有的设备和系统往往和特定的软硬件紧紧地联系在一起，很难升级和维护。开发人员常常面临不同方面的技术难题，还要重复开发类似的算法，既耗时又使成本增加。有些看似细节的问题，所涉及的处理方案可能影响整个系统，解决起来也较为复杂。用户所期待的是不用考虑产品所采用技术的不同，开发者也不希望陷入耗时费力的技术细节之中。因此，DSP 芯片的主要供应商（如 TI 公司）提供了一系列可重用的实时软件开发框架、组件、库，以及适应 SOC 的多核通信组件、编解码算法、网络开发包等。

而且，对于 DSP 工程师而言，选择一个优秀的软件开发工具将大大地加快整个开发的进度，成为帮助开发和调试的有利手段。Code Composer Studio（CCS）是 TI 公司嵌入式处理器系列的集成开发环境（IDE），也是目前使用最为广泛的 DSP 开发软件之一。CCS 以 Eclipse 开源软件框架为基础。CCS 将 Eclipse 软件框架的优点和 TI 先进的嵌入式调试功能相结合，为嵌入式开发人员提供了功能丰富的开发环境。

由以上讨论可知，现代复杂的 DSP 嵌入式系统的开发已经不再是开发人员从头开始编写所有的软件，而是以成熟的框架和算法库为基础，充分利用开发工具，才能又快又好地完成；开发人员也不是独立完成整个系统，而是分工合作，可分成算法开发人员、系统集成开发人员，以及底层驱动开发人员等。

本书的目的就是从 DSP 软件开发的各个角度阐述 TI 公司提供的 DSP 软件和开发工具，为 DSP 软件开发人员理清思路，以简化和加快 DSP 系统的软件开发。本书系统地阐述了德州仪器（TI）公司的数字信号处理器（DSP）和多核片上系统（SOC）的相关软件技术，包括可重用的软件开发框架、实时操作系统内核、算法和多媒体库，以及适应 SOC 的多核通信组件，网络开发包等。全书分为五章，第 1 章讨论 DSP 嵌入式软件开发应注意的要素；第 2 章从 XDAIS 算法标准和三种参考编程框架等方面讨论 DSP 可重用实时软件技术；第 3 章从 DSP/BIOS 实时内核、网络开发包（NDK）、设备驱动包（DDK）和多核通信组件等方面讨论多核嵌入式软件开发；第 4 章讨论了优化的 DSP 库，包括算法库、数学库、图像处

理库以及音视频编解码；第 5 章介绍了 DSP 软件开发工具——Code Composer Studio (CCS)。

本书所涉及的材料，是截止到 2014 年的最新资料。在全面整理 TI 公司相关资料的基础上，结合编者的项目开发经验，增加了实现的例子，有利于读者理解和应用。

本书是在彭启琮教授的主导下，由潘晔和廖昌俊完成的。两位编著者均完成过大量的 DSP 软硬件工程项目，对 TI 公司的 DSP 软件和开发工具十分熟悉。其中潘晔编写了第 1、2 章和第 3 章的 3.1、3.2 节，并对全书统稿；廖昌俊编写了第 4、5 章和第 3 章的 3.3、3.4 节。管庆教授提供了第 5 章的实例和重要参考意见，教研室多位研究生同学参加的项目也作为本书的应用举例，在此表示感谢。

DSP 技术发展日新月异，应用广泛，新的软件技术和开发工具层出不穷。本书选择介绍的内容难免存在不当和错误，敬请读者反馈意见和批评指正。

编著者

2015 年 11 月于电子科技大学

# 目 录

第 1 章 绪论	1
1.1 DSP 嵌入式软件开发要素	1
1.1.1 操作系统	1
1.1.2 图形化与人机交互	2
1.1.3 安全性	3
1.1.4 开发工具	4
1.1.5 代码结构	5
1.1.6 中间件和软件框架	6
1.1.7 多媒体编程	6
1.1.8 多处理器或多核 SOC	8
1.2 eXpressDSP 实时软件与开发工具简介	9
1.2.1 CCS 集成开发环境	9
1.2.2 数据可视化	11
1.2.3 操作系统方案	11
1.2.4 算法标准和框架	12
1.2.5 数字媒体软件	13
1.2.6 驱动与开发套件	13
参考文献	14
第 2 章 DSP 可重用实时软件技术	15
2.1 XDAIS 算法标准	15
2.1.1 算法标准简介	15
2.1.2 XDAIS 算法标准规则	16
2.1.3 创建符合标准的 DSP 算法	17
2.1.4 XDAIS 算法实例	23
2.2 参考编程框架	26
2.2.1 RF 简介	27
2.2.2 RF1——紧凑型编程框架	30
2.2.3 RF3——灵活型编程框架	47
2.2.4 RF5——扩展型编程框架	70
2.3 RF 应用举例——网络数字监控系统	81
2.3.1 系统框图	81

2.3.2	系统软件设计 .....	82
2.3.3	算法集成到 RF5 .....	83
2.3.4	软件流程 .....	86
参考文献 .....		90
<b>第 3 章 多核嵌入式软件开发 .....</b>		<b>92</b>
<b>3.1 DSP/BIOS 实时内核 .....</b>		<b>92</b>
3.1.1	DSP/BIOS 简介 .....	92
3.1.2	DSP/BIOS 内核 .....	93
3.1.3	DSP/BIOS 多线程程序设计 .....	101
3.1.4	DSP/BIOS 的编程和调试 .....	111
3.1.5	DSP/BIOS 线程同步 .....	129
3.1.6	DSP/BIOS 系统时钟 .....	142
<b>3.2 NDK (Network Development Kit) .....</b>		<b>146</b>
3.2.1	NDK 简介 .....	146
3.2.2	NDK 的基本架构和 API 函数 .....	146
3.2.3	NDK 应用实例 .....	153
<b>3.3 DDK (Device Driver Kit) .....</b>		<b>156</b>
3.3.1	DDK 概述 .....	156
3.3.2	DDK 的基本结构 .....	157
3.3.3	DSP/BIOS 设备驱动 .....	161
3.3.4	GIO 组件 .....	164
3.3.5	DDK 应用举例——Video Port mini - driver .....	166
<b>3.4 DSP/BIOS LINK .....</b>		<b>170</b>
3.4.1	DSP/BIOS LINK 的软件结构 .....	171
3.4.2	DSP/BIOS LINK 的关键组件 .....	172
3.4.3	典型的应用流程 .....	174
3.4.4	使用 DSP/BIOS LINK .....	179
3.4.5	应用举例 .....	181
参考文献 .....		197
<b>第 4 章 优化的 DSP 库 .....</b>		<b>199</b>
<b>4.1 DSP 的算法库 DSPLIB .....</b>		<b>199</b>
4.1.1	DSPLIB 的下载和安装 .....	199
4.1.2	利用 DSPLIB 实现 FFT 算法 .....	200
4.1.3	利用 DSPLIB 实现无限单位冲激响应 (IIR) 数字滤波器 .....	204
4.1.4	利用 DSPLIB 实现有限单位冲激响应 (FIR) 数字滤波器 .....	207
4.1.5	利用 DSPLIB 实现自适应滤波器 .....	211
<b>4.2 DSP 的数学库 MATHLIB .....</b>		<b>213</b>
4.2.1	三角函数 .....	214
4.2.2	除法函数和倒数函数 .....	215
4.2.3	平方根函数和平方根倒数函数 .....	215

4.2.4 指数函数 .....	215
4.2.5 对数函数 .....	216
4.2.6 幂指数函数 .....	217
<b>4.3 DSP 的 IQmath 数学函数库 .....</b>	<b>218</b>
4.3.1 定点算法原理 .....	218
4.3.2 如何安装 IQmath 库 .....	218
4.3.3 如何使用 IQmath 库 .....	219
4.3.4 IQmath 库的函数功能 .....	222
<b>4.4 DSP 的图像处理库 IMGLIB .....</b>	<b>229</b>
4.4.1 如何安装和调用 IMGLIB 库 .....	229
4.4.2 IMGLIB 库的函数功能 .....	230
4.4.3 IMGLIB 函数使用举例 .....	233
<b>4.5 DSP 的音频、视频和语音编解码器 .....</b>	<b>234</b>
4.5.1 视频编解码器 .....	236
4.5.2 JPEG 图像编解码器 .....	238
4.5.3 音频编解码器 .....	239
4.5.4 G.711 语音编解码器 .....	240
<b>参考文献 .....</b>	<b>241</b>
<b>第 5 章 软件开发工具 .....</b>	<b>242</b>
<b>5.1 DSP 的集成开发环境 CCS .....</b>	<b>242</b>
5.1.1 CCS 的下载和安装 .....	242
5.1.2 CCS 开发 DSP 程序流程 .....	244
<b>5.2 CCS IDE 常用工具的使用 .....</b>	<b>249</b>
5.2.1 CCS 中代码生成工具的使用 .....	249
5.2.2 CCS 中调试工具的使用 .....	255
5.2.3 CCS 中探针工具的使用 .....	260
5.2.4 图形工具的使用 .....	261
5.2.5 分析工具的使用 .....	263
<b>5.3 CCS 编程支持工具 .....</b>	<b>264</b>
5.3.1 CMD 内存定位文件的使用 .....	264
5.3.2 DSP 片级支持库 .....	275
5.3.3 DSP/BIOS 工具的使用 .....	280
5.3.4 XDC 工具的使用 .....	290
<b>5.4 C6EZ 工具的使用 .....</b>	<b>297</b>
5.4.1 C6Run 工具的使用 .....	297
5.4.2 C6Accel 工具的使用 .....	301
5.4.3 C6Flo 工具的使用 .....	308
<b>参考文献 .....</b>	<b>313</b>



# 第1章 绪 论

数字信号处理（DSP）的应用无处不在，包括消费类电子设备、工业控制、安防系统、通信系统、医疗设备、航天航空、军事装备等方方面面，而且 DSP 芯片以及多核片上系统（SOC）的功能越来越复杂，外设种类越来越多，大量新技术标准、新算法、新应用层出不穷。因此，开发一个 DSP 嵌入式系统，应以成熟的框架和算法库为基础，充分利用开发工具，才能降低开发难度和成本。

本章首先从宏观上讨论 DSP 嵌入式系统软件开发应注意的要素，然后简介 TI 公司的 eXpressDSP 实时软件组件和开发工具。

## 1.1 DSP 嵌入式软件开发要素

DSP 嵌入式系统的软件开发是一项极具挑战性的任务，需要考虑大量相互关联的要素，才能做出优化与权衡。只有事先做好规划，才能完成这一挑战；否则很容易造成开发时效率低下，甚至推倒重来。开发人员首先应明确一个基本问题：此项目的目标是什么？然而这一问题并不容易回答。明确了项目的目标意味着完全理解了影响此项目的各方面要素，最终产品就会在市场上取得成功。实际上，这些要素直接决定了开发人员的多数决策。例如，产品上市的时间和利润点决定了开发人员选用何种开发工具，是否将开源代码或商业软件集成到系统中，是否采用以及采用哪种操作系统等。下面将讨论以下几方面要素：操作系统、图形化与人机交互、安全性、开发工具与开源代码、代码结构、中间件和软件框架、多媒体编程、多处理器或多核 SOC 等。

### 1.1.1 操作系统

DSP 嵌入式系统软件开发的首要问题是考虑是否采用操作系统。有的系统很简单、功能有限，就不需要操作系统；有些微控制器也不采用操作系统，而是采用一些简单的编程接口。TI 公司为自己的 ARM 处理器 DSP 处理器提供了名为 StarterWare 的开发套件。StarterWare 是基于 C 语言的，不需要操作系统的支持；包括设备抽象层库以及外设编程的例子，例如网络、图形、USB 等。

如果需要操作系统，可选择的种类也很广泛。到底是选择购买 Windows CE，还是选择开源的 Linux，或者选择免版税但封闭的 Android？是选择高层操作系统（HLOS）还是实时操作系统（RTOS）？选择操作系统是首要任务，因为操作系统直接影响其他部分软件的开发。

#### 1. 商业 OS、开源 OS 及免版税 OS

采用商业 OS 需要向所有者付版税，购买了商业 OS 可以得到该 OS 所有者公司的技术支持和培训，从而缩短学习路线，克服不可避免的困难，加快软件开发过程。而且，由于商业

OS 完全由其所有者公司控制，第三方工具和插件等通常是经过该公司评估和认证的，可以方便地集成到大型软件系统中。

开源 OS 可以免费获得，其中 Linux 是最著名的。开发者社区为开源 OS 提供支持和插件。如果开发人员采用了开源 OS，就需要自己负责集成、评估和测试操作系统以及第三方工具和插件，以保证所有软件能在一起正确工作。采用开源 OS 的好处是可以免费从开发者社区获得创新的软件模块，许多有创造力的开发者会定期向开发者社区贡献自己的成果。另外，许多操作系统厂商，如 MontaVista、Red Hat、RidgeRun、Timesys、Wind River 等开发了商业版的 Linux。

Android 是免版税 OS，可以从 Google 免费获得，但其内容和架构仍然是 Google 控制的，例如，Android 也基于 Linux，但 Google 决定集成哪些内容和插件以及多媒体处理软件等。Android 的封闭虽然限制了开发的灵活性，但简化和无缝集成了第三方代码，从而加快了系统开发过程。

## 2. RTOS 与 HLOS

选择操作系统需要考虑的另一个重要方面是待开发的应用系统是否是实时系统。RTOS 是专门为实时系统设计的，可以满足确定的响应时间。实际上，确定的响应时间对一些实时系统至关重要，甚至关乎生死。例如，汽车的刹车系统必须在踩下刹车时立即响应，否则就会有生命危险。

通用的 HLOS 一般把命令放到队列中，然后在适当的时候再响应；而 RTOS 必须能识别紧急命令并立即响应。正是由于 RTOS 主要关注响应的实时性，通常 RTOS 不能支持类似 Windows 系统的通用 HLOS 的广泛功能；RTOS 支持的一般局限于实时系统中最主要的时间敏感性任务。

通用 HLOS 的关注重点是便于使用，故 HLOS 比 RTOS 有覆盖面更宽和更深的抽象层，以隔离用户和硬件处理器或通信系统等。而且 HLOS 通常假定用户要处理很多任务，因此 HLOS 比 RTOS 功能复杂得多，代码量也大很多。反过来，这也使得 HLOS 不适合对响应时间有确定要求的实时处理应用。

### 1.1.2 图形化与人机交互

人机交互也是 DSP 嵌入式系统软件项目需要考虑的问题。在项目的最初设计阶段，就要考虑是否需要图形化用户接口（GUI）。人机接口（HMI）或 GUI 在消费电子系统中很流行。在嵌入式系统中用户也希望有条件地采用这类直观的接口，以方便用户，提高系统的易用性以及总体操作效率。而且，最终系统在市场上是否受欢迎也与广告描述的用户接口相关。

嵌入式系统的图形处理包括三个不同的任务：创建、构图和显示。首先是生成图形元素，比如窗口、图标、按钮等；其次是设计不同窗口、桌面和其他图形元素的构图，即这些元素在屏幕上看起来是什么样子以及当系统运行时它们如何改变；最后是按某种顺序在显示器上实际显示图形元素。

显然，当用户接口越来越复杂时，图形设计和处理也越来越复杂。软件开发人员为一个特定的嵌入式系统开发用户接口时会在很大程度上依赖系统的硬件能力。例如，大部分消费电子系统配有专门的图形或多媒体处理器，而无专门图形处理器的嵌入式系统可能需要硬件



加速器,才能分担系统主处理器的图形处理运算量,从而改善系统的图形响应性能。用户接口会影响用户对系统整体响应时间的感受,也就是说,一个反应迟钝的用户接口会给用户造成整个系统运行糟糕的印象。

对软件设计人员而言,一个需要考虑的关键问题是,一个特别的 HMI 或 GUI 对于系统在市面上的成功应用是否是绝对必要的。当然,用户接口对于许多消费电子设备或系统获得成功起了重要作用,但对于嵌入式系统就不是那么重要了。嵌入式系统的用户接口设计者需要在多方面进行权衡,分析各种图形和显示方案隐含的成本,确定满足系统成本、功能和性能的最优组合。另外需要考虑的问题包括:自行开发用户接口还是外聘专业的开发人员;需要多大的存储空间等。

现在已经有很大图形界面开发工具,包括开源的和商业套件。OpenGL 是一种应用广泛的跨平台图形框架,独立于任何 OS,由 khronos.org 提供支持。另外,有很多 Linux 平台的图形开发框架,如:Qt、X-11、GTK 和 DirectFB,每种框架都有多种实现子集,以及不同层次的硬件抽象。由于 Linux 本身不包括这些图形框架,因此对于没有经验的开发者,要在一种特定的嵌入式处理器上集成并运行 Linux 和这些图形框架是很耗费时间的。除了开源软件,一些商业 OS 也包括图形框架,如 WindowsCE、VxWorks、QNX 等;还有独立于 OS 的图形框架 Mentor Graphics' Inflexion,既可以运行于开源 OS 也可以运行于封闭 OS。

### 1.1.3 安全性

嵌入式系统如何考虑安全性?还是完全不需要关注安全方面?实际上,在面对黑客、病毒以及恶意程序时,电子设备是很脆弱的。在可预见的将来,安全性将是业界持续关注的主要问题。存储了用户私有信息的嵌入式系统不会对身份盗窃者、篡改者、恶意攻击以及匿名恶作剧免疫,而如医疗系统和紧急应答通信这些类型的嵌入式系统对个人和公众安全十分关键。因此,嵌入式开发人员必须在项目刚开始就做好计划以保护系统,即通过设计适当的安全措施来保护操作和存储的信息。

对于保护嵌入式设备,理解黑客的目的是极重要的。许多设备远离安全设施,放在很容易受攻击的位置,更需要强健有效的安全措施。开发人员思考以下问题有助于集中注意力,部署和开发适当的方法为最关键的部分提供足够的保护。

- 什么是需要保护的?
- 谁是安全措施要对抗的?
- 如果试图保护的信息被泄露了,代价是什么?

当然,这些问题的答案会根据最终应用发生变化。例如,销售点终端要保护存储在终端上的以及通过连接到终端的通信信道发送的信用卡信息;媒体播放器要保护包含数字版权的内容的安全以免其被盗版;语音和数据通信终端要保护个人身份不被盗窃以免隐私被非法侵犯。深入回答这些问题会使开发人员的决定全面而有效,开发人员会找到什么数据必须保密以及哪些通信信道必须加密。系统中存在的第三方应用程序也必须被评估,必须充分讨论其可能的安全隐患。

既然不同的系统要求不同级别的安全性,在一些系统中经常采用的安全技术在另外的系统中却不采用。例如,在大众消费应用中,密码加速、安全启动和保护系统调试通道等构成了最低程度的安全保护,而在嵌入式应用中需要另外的安全措施来保护运行时环境以避免窜

改电压和时钟，以及保护各种通信信道。

### 1.1.4 开发工具

通常处理器供应商会提供一些开发工具和资源，包括：集成开发环境（IDE）、硬件评估模块/开发板（EVM）、软件开发包（SDK）等。这些工具为项目初期提供了测试平台，可以极大地减少开发时间。

IDE 在业界很流行，目前市场上有开源的、商业的以及处理器供应商提供的各种 IDE。IDE 将多种工具集成在同一环境中，可以简化嵌入式软件开发。在 IDE 中，多种开发工具通常共享相同的用户接口、命名方式、命令集以及其他总体操作等，数据在开发工具之间的搬移也较容易和直观。然而，并不是所有的 IDE 都是类似的，可能相当部分的 IDE 只有基本的代码生成工具，如汇编或编译器。而其他开发工具没有紧密地集成在一起。于是工具间的转换很费时，会影响整个软件开发过程的效率。

另外，IDE 的性能也很重要。某些基准程序可以作为这方面的比较基础，或者通过不同的 IDE 运行大量已有代码也可以进行 IDE 性能比较。而且，不同 IDE 提供的可视性区别也很大，良好的可视性能提高开发和调试代码的效率。

有些处理器供应商的 IDE 以开源的 Eclipse 为基础。Eclipse 可以支持多种编程语言，如 Java、Ada、C、C++、COBOL、Python、Ruby 等。Google 公司根据 Eclipse 开发了 Android 的开发工具，TI 公司在 Eclipse 的基础上开发了自己处理器的 IDE——CCS。于是采用 TI 处理器，基于 Android 平台的嵌入式系统可以得到 Google 和 TI 的双重支持。

多数 EVM 包括一块验证过的印刷电路板（PCB）和相关电子元件，以及少量软件资源，不过不同 EVM 支持的软件资源变化很大。支持广泛软件资源的 EVM 可使开发人员更快地搭建好一个应用实例。更完整的 EVM 可以支持 IDE 中的 SDK、代码库，以及 Linux、Android 或 Windows CE 这类操作系统。

SDK 通常由芯片厂商免费提供，其基本组件通常为特定的代码生成器，但有的 SDK 包括了一个特定应用的全部软件参考设计。后面这种类型的 SDK 中，构成系统的大部分软件都已经准备好了，直接可以使用，例如：多媒体框架、编解码器、控制 USB 和网络等接口的输入输出代码。这样，软件开发人员可以集中精力实现系统的上层应用软件。然而，这类包含广泛软件的 SDK 的坏处是它会限制开发人员实现与众不同的创新功能。因为这类 SDK 的大部分软件为二进制目标码，开发人员无法访问底层源代码，从而很难进行个性化修改。反之，那些由源代码组成的 SDK 给开发者提供了修改和定制的机会。只要许可协议允许，开发者就可以按自己意愿进行修改。

SDK 除了要易于定制修改，还应提供结构化的、文档清楚的应用编程接口（API）。一个有效的 API 能简化编程，有利于软件构架的重新配置。API 还应为系统软件的每层提供钩子（HOOK），以便开发者能方便地将自己的软件模块加入到系统中，而不用重新编写整个新模块。这样可以显著减少软件开发时间，加快新产品的上市。

处理器供应商对开源软件的支持很重要。开发人员选择开源软件时要考虑是否能方便地移植到特定的处理器，若处理器供应商已经完成了移植更好。例如 TI 公司，已经将大量的开源中间件和工具移植到了自己的处理器上，以便开发人员直接使用。这些开源软件包括：

➤ OpenMax；

- GStreamer;
- Eclipse IDE;
- OpenGL;
- OpenCV;
- Yocto;
- Linaro。

TI 为嵌入式系统提供了移植好的 Android 软件库，包括操作系统、中间件以及其他工具，适用的处理器包括 TI 的 ARM、DSP + ARM、DSP、OMAP 以及数字媒体处理器等。详见 [arowboat.org](http://arowboat.org)。

### 1.1.5 代码结构

代码结构对于嵌入式软件编程是很重要的。在设计软件时应考虑代码结构，这会影响到将来代码的重用、个性化的设计，以及是否容易调试等。

#### 1. 代码的重用

若在不同的应用中，代码、模块，甚至整个软件系统可重用，则软件开发的投资回报就会增加。为了取得最大的投资回报，制造商在制定产品线策略时，希望尽可能多的软件能在所有产品中重用。因此，代码重用成为了软件开发团队需要首先考虑的问题。

理论上，在一个产品线中，每一次提升系统能力常常意味着提高处理器性能，或者更换更强大的处理器。为了确保软件重用，软件团队必须考虑软件对产品线中不同处理器的兼容性，理想情况是为一种处理器开发的软件可以直接下载到产品线的其他系统中。因此，系统设计者必须仔细考虑各种处理器对软件的兼容性，以及软件对不同处理器的可移植性。

开发工具能帮助实现跨处理器平台的软件重用。例如，C 等高级语言编译器允许一定程度的代码重用。具有良好结构和完整文档的 API 也能保证软件在多种平台和处理器之间的扩展性。然而，有些 API 相对于它们的代码而言过于庞大臃肿，从而对系统性能造成了负面影响。

总之，当软件团队具有广泛的开发工具、软件组件、封装器，以及其他能力，就可以实现高层软件重用；即可以很容易地混合以前开发的代码、加入或裁减掉某些特性和功能，从而快速地实现待上市产品的特定需求。

#### 2. 代码的定制

许多软件开发项目的一个重要目的是为最终产品提供独特的性能或与众不同的特性。这样，一旦产品上市，就能从竞争中脱颖而出，从而取得极大成功。软件开发团队在开发新的特别功能，并将他们与开源软件集成时，必须特别小心。除非将自己的软件系统严格组织，并与开源软件完全隔离，否则根据开源软件许可协议，这些新开发的独特软件也会被认为是开源的。因此，必须将这些代码贡献给开源社区。

一些技术公司已经采取了一些步骤来预防这类情况。例如，TI 公司提供的软件附有一份“软件清单”，详细解释和描述了这些代码的来源，以及相关的、需遵循的任何第三方或开源软件许可证。而且，TI 公司的开源软件审查委员会已经检查过了这些软件清单以及代码本身，以确保这些代码具有严格的组织和架构，从而保护系统厂商的利益。

#### 3. 调试

几乎没有软件项目可以不经过调试就能发现和消除无法预料的错误和故障。事实上，如

果软件设计成便于调试，可以更快地实现全部功能。搭建软件系统的架构与各独立处理节点的组合时，可以考虑尽可能多地为调试提供代码可见性，以便于软件开发人员孤立错误、快速修复有问题的代码段。

处理器供应商也会提供基于硬件仿真器的调试工具。例如，TI 提供全系列硬件仿真器，可以在处理器运行时查看相关的状态。而且，这些仿真功能是 TI 的 IDE——CCS 的组成部分，能方便用户在源代码中快速定位并修复故障。

### 1.1.6 中间件和软件框架

已经有大量的资源可以供嵌入式软件开发人员使用。当然，选择特定操作系统和操作系统类型将影响贯穿项目始终的各种软件资源，包括中间件、软件框架，以及其他工具软件。例如，与类似 Windows CE 的商业操作系统相关的软件资源通常受操作系统供应商或授权的第三方厂商的限制；反之，选择类似 Linux 的开源操作系统的开发人员必须从开源社区提供的广泛的开源中间件和工具中评估和选择自己需要的软件资源。而且，项目组可能不得不在自己的硬件上移植开源软件资源。不过 TI 等一些硬件厂商已经为用户完成了很多开源软件资源的移植。随着基于 Linux 的免版税 Android 系统的推出，多数开源软件资源提供了基于 Android 的版本。另外，TI 等芯片厂商以及其他技术公司也提供了专有的软件开发资源。

现在，软件开发人员使用开源中间件越来越普遍。有些中间件，如 GStreamer、Open CV，以及 OpenMax 等已经在某种程度上遍布业界。开源的工具链，如 Yocto 和 Linaro 等也已崭露头角。

中间件是一组预先定义并完全开发的软件功能或任务，开发人员可以选择其中的一些组件来部署自己的系统。软件框架为集成各子任务确定了一组规则，以便减少集成的时间和难度。典型的中间件是实现特定类型的处理任务。如 GStreamer 专注于多媒体处理，需要部署多媒体处理栈的开发团队可以从其中选择多种模块，如视频解码模块或音频处理插件。

TI 的微控制器和微处理器已经提供了许多移植了各种中间件的软件开发套件。这样可以使项目开发简化步骤，不用为硬件平台移植中间件，而是直接在最流行的中间件基础上集成自己的任务。

### 1.1.7 多媒体编程

一般而言，多媒体编程包括处理视频、音频、图像以及图形。一些主要考虑的问题包括支持何种编解码器、需求哪些 I/O 接口、系统实时性的要求、系统并发性的需求、要支持几个通道、比特率和帧率、显示分辨率、多媒体内容的质量、存储容量和带宽等，所有这些方面都会影响多媒体编程。

许多系统处理视频和音频内容并输出到音/视频端口。处理器厂商也许会提供音频和视频驱动，但嵌入式开发人员仍需要熟悉各种音/视频接口，以便出现问题时进行调试。音频接口有模拟和数字形式。对于模拟音频接口，系统中要包括模数转换器（ADC），数模转换器（DAC）或二者的结合（codec）。对于数字音频接口，通常为 SPDIF 和 HDMI 接口。模拟视频接口包括射频（RF）、复合视频、分离视频等。这些模拟视频流通过视频 ADC 构成的解码器转换成数字形式。视频编码器则将数字视频转换成模拟视频流供监视器显示。最常用



的数字视频接口为 HDMI、DVI 和 LCD。一些系统会与相机接口，包括 CMOS 传感器、智能传感器，或 USB 相机接口。这些接口有时会集成到一片 SOC 器件上。若没有集成到 SOC，系统开发者就必须选择不同的部件来实现要求的接口，此时必须确保所选的部件包括了合适的驱动。

许多嵌入式系统必须能处理压缩的音频和视频数据。这时，系统设计者和编程人员需要考虑如何实现压缩和解压。系统可能输入各种压缩的媒体数据，包括自己的文件系统从硬盘、USB 闪存或 SD 卡读取的压缩内容，从有线或无线网络下载的压缩比特流等。通常，系统软件必须能解析特定的数据头，以便获知相应的压缩算法。当前常用的主流媒体格式有：数据报流（TS）、程序流（PS）、AVI、MPEG-4、ASF 等。处理数据头的软件模块的主要功能之一是对音频和视频数据解复用，对比特流解复用后，将压缩数据的缓冲区送到解码器或解压器。表 1.1 列出了一些主流编解码器。

表 1.1 主流多媒体编解码器

媒体类型	编解码器
视频	H.264 (AVC)、MPEG-4、MPEG-2、VC1、VP6/7/8、RealVideo
音频	AAC、MP3、WMA、MLP、AC3、Vorbis
图像	JPEG、PNG、GIF、TIFF
语音	G.7xx

### 1. 存储容量问题

目前许多嵌入式系统采用 DDR 存储器。为了降低系统成本，设计人员要在存储成本和容量之间进行折中。也就是说，设计者希望用最低成本的存储器提供能满足系统多媒体处理要求的足够带宽。

视频处理通常消耗大量的 DDR 容量，而音频处理消耗的相对少一点。将视频流传输到输入输出接口、解码、编码、后处理以及显示视频均会消耗大量存储器。编程人员应该意识到，应避免高清晰度视频的缓冲区拷贝，因为这种操作会超过系统中 DDR 的容量限制，造成低视频帧率或显示异常，从而给用户很差的使用感受。

### 2. 图形/视频混合

在许多应用中，图形和视频必须混合或组合以呈现给用户显示。一些嵌入式处理器包括了显示子系统，能直接用硬件进行这种混合。若没有显示子系统，需要用二维图形引擎来混合视频和图形。有的应用要实现三维（3D）视频，于是需要三维图形引擎。各种图形中间件也能帮助实现图形与视频的混合。

### 3. 响应时间

响应时间是指系统给出用户响应的时间。对最终用户而言，系统的响应时间至关重要。例如，如果视频会议的响应时间过长，图像或声音通道就有延迟，会对会议的参与者造成混乱。一种减少响应时间的方法是处理流水时减少缓冲区的大小，而且流水的每一步都要严格检查和优化以满足要求的响应时间。

### 4. 音视频同步

在音频/视频内容解码时，要求同步音频和视频流。例如，达到口型同步很常见，更复杂的同步要求系统的音频/视频解码与广播源等编码器同步。这种与源编码器同步的要求很常见，可以保证不跳帧或重复，从而减少运行时的比特流缓冲，避免溢出。

## 5. 质量

不同的应用要求不同程度的媒体质量。要达到高质量的音/视频，通常涉及软件算法或加速器，会消耗处理资源。因此，系统设计者必须考虑为多媒体处理任务分配足够的资源。而且，编程人员已经发现在代码的关键点设置内嵌测试能帮助调试质量问题。

## 6. 多媒体框架

OpenMax 和 GStreamer 都是免版税的跨平台多媒体中间件，软件开发人员可以快速将媒体处理模块插入到自己的系统中。这些多媒体模块包括音频、视频、静态图片以及其他类型的媒体处理例程。这些例程通常用 C 或 C++ 等高级语言编写，包括为应用程序提供抽象层的编程接口。

OpenMax 由 Khronos 发起，赞助商有 Evans&Sutherland、ARM、TI、NVIDIA、SGI、Oracle/SUN 等。GStreamer 由 Oregon Graduate 于 1999 年发起，已经被 TI、Nokia 等公司采用。

### 1.1.8 多处理器或多核 SOC

多处理（MP）通常是指一种系统设计方法，特点是在多处理器或多核上运行多线程软件。当前构建计算或通信系统时，MP 流行的原因在于要克服单处理器结构的局限性。单处理器系统受限于处理器的速度，而多处理系统可以通过部署多处理器或处理引擎提高系统的吞吐量。除了性能的提高，与基于单个复杂、庞大的处理器系统相比，MP 结构还可以降低功耗；而功耗对于采用电池的移动设备而言十分重要。

由于 MP 的明显优势，很多嵌入式系统转向采用此技术。然而，如实时处理等嵌入式应用的典型特性可能无法在多数通用系统中找到。因此，嵌入式系统要特别考虑软件开发环境和工具是否支持嵌入式多处理。

自然地，MP 结构要求重点考虑处理器间的通信和协调以及组成多线程软件的各线程。在硬件方面，必须采用一种互联方法，而这决定了系统通信模式是采用消息传递机制还是共享内存机制。消息传递机制为处理单元配置私有存储空间，组成分布式多处理系统，数据采用消息包的形式传递。共享内存机制为所有的处理器设置一大块共享内存空间，数据通信就是将其放入共享内存，这种机制需要考虑缓存的一致性、存储单元的一致性以及同步原语。

从嵌入式软件开发的角度的角度，MP 结构会涉及在处理器之间进行负载分配，如何协调系统中的处理器，如何将多线程软件分给各处理单元以及其他问题。幸运的是，有开发工具和工具集可以提供帮助。

#### 1. OpenMP

OpenMP 是共享内存并行编程开发环境事实上的标准。TI 为它提供技术支持，而且已经移植到了许多的 TI 处理器上。OpenMP 是一个高层编程结构，以便显著简化开发多线程软件的许多问题。使用 OpenMP，用户为一个程序具体化并发策略时，只需按照编译器的指引，在高层代码标注，说明一段代码会被哪些线程运行，而编译器完成代码到处理单元的详细映射。OpenMP 可以运行于实时操作系统。

#### 2. EZ tools

大量的嵌入式系统采用异构多处理结构，即一个或多个通用处理单元，以及若干数字信号处理器（DSP）。DSP 处理单元通常加速如编解码等处理密集性任务。TI 的 EZ tools 套件

简化了嵌入式软件开发的任务，可以在通用处理器和 TI C6000 DSP 之间分割代码。

EZ tools 套件由三个不同的工具组成：EZFlo、EZRun 和 EZAccel。每个工具针对不同的开发需求。EZFlo 面向 DSP 开发者，它有直观的图形化用户界面，允许拖放以及连接快速代码生成和应用原型的功能块。EZRun 面向 ARM/Linux 开发者，便于移植 C 代码为 ARM 的可执行代码或库。EZAccel 面向要实现 TI 编解码框架的系统开发者，它可以加速将 DSP 函数加入编解码器。

## 1.2 eXpressDSP 实时软件与开发工具简介

TI 公司提供了 eXpressDSP 实时软件和开发工具的组合，以帮助用户完成 DSP 嵌入式系统的软件开发。组合中包括一系列紧密连接的部分，使开发者能充分发挥 TI 的 TMS320 系列 DSP、Davinci 以及 OMAP 等处理器的潜力。每部分的设计都是为了简化编程，以及将定制软件开发方法转换成一种新的协作模式，即从广泛的多家厂商寻求支持。随着实时应用的爆炸性增长，对数字信号处理器的实时处理能力的需求也随之增长。eXpressDSP 可以使创新者和发明者加快新产品投放市场的速度，让思想的火花变成现实。

图 1.1 描述了 ARM + DSP 架构上的 eXpressDSP 实时软件的组件以及在主机上的开发工具。包括 CCS 集成开发环境、数据可视化工具、操作系统、算法标准和框架、数字媒体软件、驱动与开发套件等。

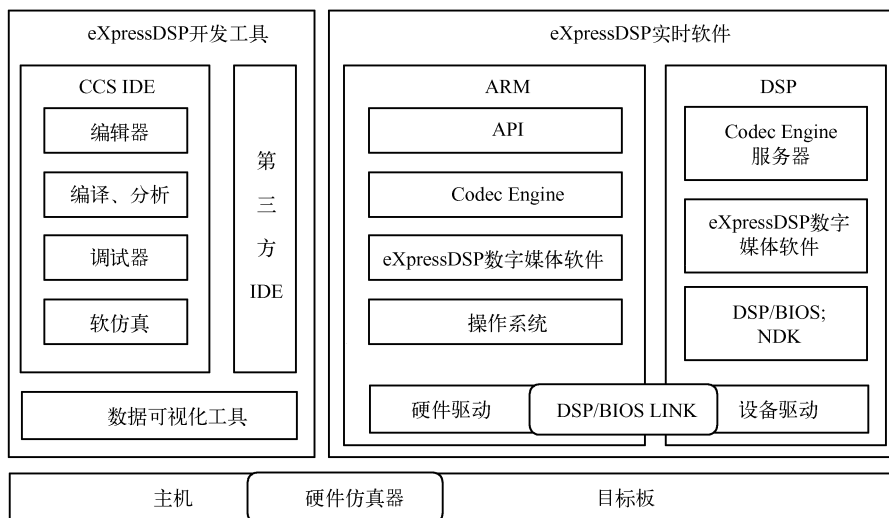


图 1.1 eXpressDSP 实时软件与开发工具

### 1.2.1 CCS 集成开发环境

CCS 是一个完整的 DSP 集成开发环境（图 1.2），也是目前使用最为广泛的 DSP 开发软件之一，支持所有的 TI DSP，包括 TMS320C6000、TMS320C5000、TMS320C2000、DaVinci、OMAP 等处理器。

## 1. 集成可视化开发环境

CCS 提供了集成开发环境，使整个 DSP 软件开发流程都可以在统一界面中完成。对用户而言，熟悉的工具和界面会降低学习难度，更快地上手。在可视化代码编辑界面（Editor），用户可以方便地编写 C、汇编、.H 文件、.cmd 文件等。使用内建的工程/项目管理器（Project Manager），可以很容易地管理多用户和多种类项目。

提供 GEL 工具，用户可以编写自己的控制面板/菜单，方便地修改变量和配置参数等。

开放式的 plug-ins 技术，支持其他第三方插件，支持包括软仿真在内的各种仿真器。

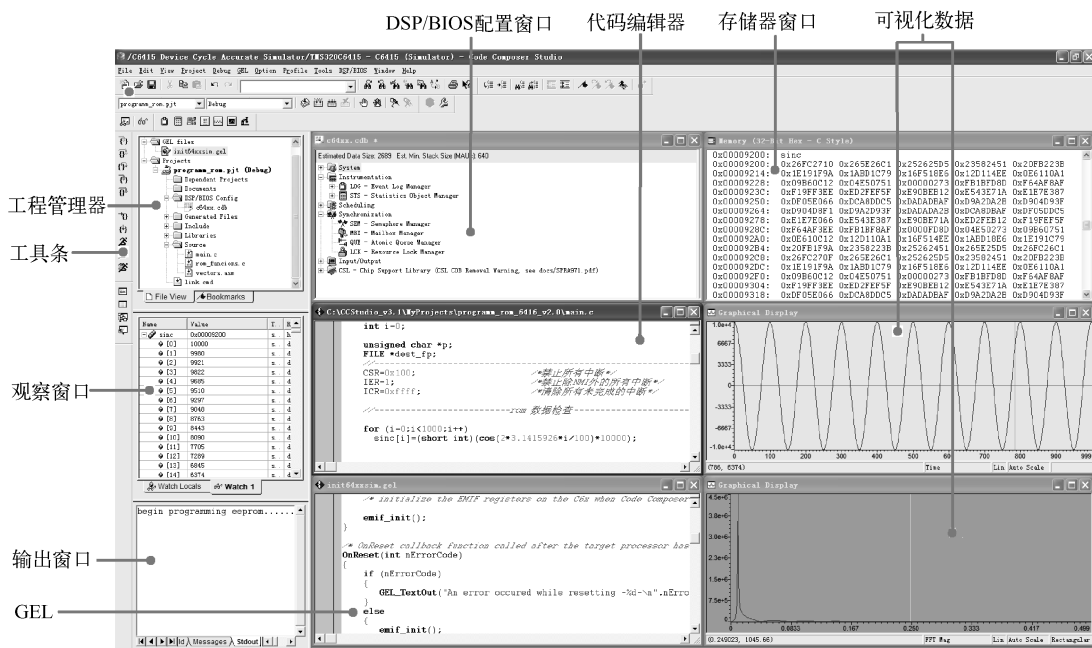


图 1.2 CCS 集成开发环境

## 2. 代码生成和分析工具

代码生成工具包括汇编器、优化 C 编译器、连接器等。以前开发高性能 DSP 程序需要清楚特定器件的结构，并手工优化汇编代码；但现在 TI 的 C 编译器可以帮我们完成优化。编译器评估代码的性能，设置了多种优化级别，可以生成小而且快的汇编代码。

分析工具（profiler）用于评估代码的性能。CCS 集成的分析工具可以方便地分析所有 C 或 C++ 函数的指令周期数、缓存的命中率、流水延误与分支等。这些分析结果对于代码优化很重要，可以明确需优化的重点代码段。

## 3. 调试器

CCS 集成了强大的调试工具，支持 C 源代码级调试以及多 DSP 和 ARM/DSP 联合调试。基本调试工具有装入可执行代码，查看寄存器、存储器、反汇编、变量窗口（Watch Window）等；断点工具，包括硬件断点、数据空间读/写断点、条件断点等；还支持对多个处理器起作用的全局断点。TI 的器件包括先进的硬件仿真特性，可以支持实时调试。当运行时间敏感的中断服务程序时，仍然允许 CCS 查看寄存器和存储器。

CCS 提供了探针工具，可用于算法仿真和数据监视等。先进的事件触发功能，可以设置为：当一系列复杂事件按时序发生时让处理器暂停。数据的图形分析工具可绘制时域/频域



波形、眼图、星座图、图像等，并可自动刷新（使用 Animate 命令运行）。

#### 4. 软件仿真器 (Simulator)

软件仿真器提供了一种不用硬件电路板运行 DSP 程序的方法，可以调试算法功能。有一些软件仿真器还可以进一步地验证时钟周期、运算速度以及部分外设功能。

软件仿真可以帮助用户跟踪和查找程序的问题。Watch 窗口可以帮助找出内存冲突问题。中断响应时间探测器可以测出响应时间最坏的情况。TMS320C55x 软件仿真器的流水分析通过显示流水的细节，分析延误及其原因。代码覆盖检查报告各源代码段（C 和汇编）是否会执行。

#### 5. 实时数据交换 (RTDX)

RTDX 和高速 RTDX 可以在不中断目标系统运行的情况下，实现目标系统与主机交换数据，即在目标系统与主机间提供了一个双向数据管道。开发者可以从目标系统存取数据，以便实时观察，仿真数据输入，缩短开发和调试的时间。

### 1.2.2 数据可视化

实时显示数据是优化和调试系统的关键。硬件仿真器提供了目标系统与主机的链接，使主机可以控制目标系统。eXpressDSP 数据可视化工具可以以图形方式显示复杂的数据集。

XDS560 是高速、先进的硬件仿真器。它的跟踪模块结合数据可视化，可以帮助用户找出很难发现的复杂实时故障，例如：事件间的竞争条件、栈溢出造成的程序崩溃、失控代码、虚假中断等。这种跟踪是完全非介入式的，它依赖于 DSP 芯片中的调试单元，完全不会影响或改变目标系统的实时运行行为。

eXpressDSP 数据可视化工具还提供了 SOC 分析器。SOC 分析器属于 TI 的高层系统调节和显示工具，能使开发者动态观察 SOC 流数据，而不是只能事后分析静态数据。通过捕捉和图形显示系统的相互影响与负载分配，隔离瓶颈，确认异常行为，以及检测性能，开发者能极大提高效率 and 总体性能，避免烦琐的手工数据的收集和比较。

### 1.2.3 操作系统方案

TI 公司为自己的 ARM 和 DSP 处理器提供了全面的操作系统解决方案，包括 DSP/BIOS、Linux、Windows CE 等。

#### 1. DSP/BIOS

DSP/BIOS 实时内核随着 CCS 集成开发环境一起发布，免版税，支持 TI 公司全系列 DSP 芯片。DSP/BIOS 是一个多线程实时内核，其健壮性已经经过了数千个嵌入式系统的验证。DSP/BIOS 是高度可裁剪的内核，可以根据需要生成最小的可执行代码；也可以扩展网络开发套件（NDK）以及与 ARM 连接和通信的 DSP LINK 模块。

DSP/BIOS 内核提供抢占式多线程，缓存和中断管理，邮箱、旗语和变长消息等通信机制可以无差别地工作于单核和多核系统中。为了调试复杂的应用系统，DSP/BIOS 包括了实时记录服务，与 CCS 的图形分析和显示工具集成，可以增强对代码的实时分析能力，如线程的执行顺序、CPU 负载、系统资源的使用情况等。

DSP/BIOS 常运行于 DaVinci、OMAP 等多核处理器中的 DSP 核，这时，DSP 需要通过

DSP LINK 模块与 ARM 端操作系统（Linux 或 Windows CE）通信。网络开发套件提供标准的 TCP/IP 网络服务，并包括 HTTP、TELNET、DNS 等高层应用协议。

## 2. Linux

TI 赞助了 OMAP 和 DaVinci 的 Linux 社区，为自己的处理器提供开源 Linux 的最新裁剪版本，用户可以免费下载。开发工具为 G++，包括 GNU 的 C 和 C++ 编译器以及 Eclipse IDE。支持 TI 的 ARM 核处理器，包括 OMAP 系列。主机端为 Linux 或 Windows，目标系统可以为 GNU/Linux、uClinux 或 EABI。

对于希望采用 Linux 操作系统，又能得到商业公司支持的用户，TI 提供了合作伙伴 MontaVista 的专业版 Linux。此商业版本 OS 支持 DaVinci 和 OMAP 处理器，在主机端提供了编译、链接、调试、代码分析等工具；在目标板上提供了全部驱动和丰富的软件组件。MontaVista 专业版 Linux 由 MontaVista 公司提供完全的技术支持和维护，使用许可可以作为 TI 数字视频软件包（DVSPB）的一部分来购买。

## 3. Windows CE

嵌入式 Windows CE 是一个组件式的实时操作系统，广泛用于小型设备，例如：手持 GPS、工业控制等。TI 为 OMAP35x 和 TMS320DM644x 等处理器提供了 Windows CE 的数字视频开发包（DVSDK）。

OMAP35x 可用嵌入式 Windows CE 6.0，开发工具为 VC2005 IDE 中添加一个插件。由于有很多测试过的操作系统组件和熟悉的开发工具，嵌入式 Windows CE 6.0 具有灵活易用、安全，以及兼容性好等优点。

### 1.2.4 算法标准和框架

TI 的多媒体框架包括 xDAIS 和 xDM 算法标准，以及 Codec Engine 算法执行框架。xDAIS 和 xDM 算法标准可以简化将多个算法集成到目标系统的过程。框架组件能使用户采用符合算法标准的算法快速构建系统。

#### 1. 算法标准

xDAIS 是 eXpressDSP 算法互操作性标准，xDM 是 eXpressDSP 数字多媒体标准。xDAIS 要求算法让应用框架决定如何分配资源，消除了算法直接访问硬件资源造成的集成问题，提高了时间回报率。

xDM 将通用的 API 具体化为针对特定的算法集，使能了一个集成器，以便功能或性能要求改变时能快速改变算法。xDM 主要定义了视频和音频编解码相关的 API。

#### 2. 框架组件

对于希望开发与 eXpressDSP 兼容的框架的开发者，TI 提供了底层组件，如 DSKT、DMAN3 等。这样就可以查询算法的内存和 DMA 资源的分配情况。

#### 3. Codec Engine

Codec Engine 是一个算法执行框架，自动调用 eXpressDSP 兼容的算法实例。它可以在 ARM、DSP 或 ARM + DSP 环境中执行，支持并发执行多通道和多算法。

Codec Engine 的设计目的是用于连接提供音频/视频同步、I/O 或网络服务等功能的高层框架或中间件，从而使系统供应商很容易实现差异性的应用系统。

Codec Engine 的一个重要特性是在跨平台上（TI SOC 与 DSP）保持 API 的一致性，于是

用户能在多种 TI 的处理器上灵活裁剪和移植程序。

### 1.2.5 数字媒体软件

eXpressDSP 数字媒体软件是一组经过实际产品测试的编码器、解码器、编解码器以及最常用媒体处理函数库，并在 TI 的 DSP 和 SOC 平台上进行了优化。这样，设备开发商就不用投入时间和精力到标准媒体编解码和处理库的实现，从而节约了数年的时间，可以专心做好具有自己特色的应用系统。用户可以在购买 TI 的 EVM 套件时得到 eXpressDSP 数字媒体软件的评估版本，另外还有多种灵活的使用许可可以适应任何需求。关于数字媒体软件的详细清单以及适合的硬件平台请参考：[www.ti.com/digitalmediasoftware](http://www.ti.com/digitalmediasoftware)。

eXpressDSP 数字媒体软件与简单例子软件和自由软件不同，其全部组件都遵循严格编码规范，包括数据手册、版本说明、用户指南和使用举例等完整文档。每个模块均包括针对目标板的可重入代码库；都经过了单元测试和系统测试，以及数千厂商在世界各测试实验室进行了测试。这些编解码器已广泛地用于终端设备和系统，包括无线手持设备、网络设备、视频或 IP 电话、流媒体设备、机顶盒、视频会议等。

eXpressDSP 数字媒体软件集是由 TI 公司以及 TI 授权的软件供应商提供许可，每个组件都与 xDAIS 和 xDM 算法接口标准兼容，很容易与 eXpressDSP 软件框架集成。其全部软件集由 TI 授权的软件供应商提供高质量的技术支持和培训，方便用户快速应用到自己的系统中。购买请参考：[www.ti.com/codecbundles](http://www.ti.com/codecbundles)，TI 授权的软件供应商列表见 [www.ti.com/asp](http://www.ti.com/asp)。

### 1.2.6 驱动与开发套件

TI 公司与 TI DSP 开发者网络提供了广泛的硬件和软件入门开发套件，以及产品级的解决方案。这些给用户提供了很好的方法来评估 TI 的处理器和数字编解码器，并测试驱动，eXpressDSP 工具为开发应用系统提供了很好的开始。很多产品级的解决方案包括了完整的软件许可证、开发好的软件，以及独立的硬件仿真器。另有大量的子板可以增加用户需要的功能。参考设计为用户开发硬件和软件提供了参考，以便快速实现产品。设备驱动开发包 (DDK) 可以帮助开发者实现自己需要的驱动。

#### 1. DSP 入门套件 (DSK)

DSK 提供了低成本的入门机开发套件，可以评估 TI 处理器平台以及 eXpressDSP 开发工具。TI 及其合作伙伴为每种处理器提供了多种 DSK，让开发者很容易学习和使用。可以测试算法、验证设计、评估代码和处理器架构等。DSK 与能运行于特定目标板的 CCS IDE 一起发售，通常 DSK 板上集成了仿真器。目前 TI 提供了针对 TMS320C2000、TMS320C5000，以及 TMS320C6000 处理器平台的多种 DSK。

#### 2. 数字视频开发套件

广泛的 ARM/DSP 系统工具帮助开发者跳过数字视频开发的起始阶段，简化复杂应用的开发过程。这些开发工具有：数字视频开发平台 (DVDP)、数字视频评估模块 (DVEVM)，以及数字视频软件开发包 (DVSPB)。

DVDP 是针对 DM648 和 DM6437 的数字视频开发平台，包括：DM6437 或 DM648 处理器目标板；eXpressDSP 基于 DSP 的数字视频开发套件，具有 TI 的实时 DSP/BIOS 核、codec

engine、视频编解码器、音频编解码器、芯片支持库 (CSL), 以及基于 DSP/BIOS 核的器件驱动器; 可运行于 DM6437 的虚拟 Logix Linux 核; 可运行于 DM648 的 Ittiam 视频编解码器演示软件; CCS IDE; 数据可视化 SOC 分析器; 以及网络开发者套件 (NDK)。

DVEVM 包括硬件和软件, 使开发者可以立即开始评估基于 ARM9 的 DaVinci 处理器, 并开始建立数字视频应用程序。目前可提供 DM6446、DM355 和 DM6467 的 DVEVM。硬件包括: 基于 ARM9 的处理器器的开发板、遥控器、视频摄像机/LCD、大容量存储器、连接电缆等。软件包括: eXpressDSP 的 Linux DVSDK、支持开发板的软件包和驱动器、Green Hills 的 Multi IDE 评估工具 (目前只有 DM6446 的版本) 等。

DVSPB 是 Linux 软件开发包, 用来快速而有效地开发基于 ARM9 的 DaVinci 处理器的复杂系统。DVSPB 将 eXpressDSP Linux DVSDK 与 MontaVista Pro Linux OS 组合在一起, 显著提高了软件的集成度和可视性。MontaVista Linux 操作系统已经针对数字视频应用和 ARM9 处理器做了优化。DVSPB 需要与 DVEVM 或 TI 的第三方硬件平台一起使用, 其中还包括了 MontaVista Pro Linux 的使用许可, 一年访问 MVZone, 以及 Dev Rocket IDE; TMS320C6000 Linux 编译器; eXpressDSP 数据可视化工具; 以及 CCS 与 XDS560R JTAG 仿真器。

### 3. DSP/BIOS LINK

DSP/BIOS LINK 是服务于通用处理器 (GPP) 与 DSP 之间通信的基础软件, 提供通用的 API 对 GPP 与 DSP 的物理连接进行抽象。可以跨平台使用, 既可以用于 SOC 内部的 GPP 和其中一个或多个 DSP 通信, 也可以用于独立的 GPP 和一个或多个 DSP 通信。

DSP/BIOS 运行在 DSP 上, GPP 上不要求特定的操作系统。DSP/BIOS LINK 是免版税的, 以两种形式提供: 移植到 Linux 上的版本或一个通用的可移植软件包。可以从 [www.ti.com/dspbioslink](http://www.ti.com/dspbioslink) 免费下载。

根据其支持的硬件平台和操作系统, DSP/BIOS LINK 提供以下服务。

- 基本的处理器控制: GPP 装载 DSP 程序, 启动或停止 DSP, 允许 GPP 访问 DSP 的资源, 用户事件的验证等。
- 跨多个处理器的共享/同步存储器池, 对共享数据结构的访问。
- 多个处理器的通信协议, 以及不同类型的数据传输协议。
- 底层同步与阻塞。
- 数据传输方式: 消息队列 (MSGQ)、基于数据流的环形缓存 (RingIO)、逻辑通道 (CHNL)。

## 参 考 文 献

- [1] Jason Kridner, Nick Lethaby, Steven Magee, Erik Welsh, Henry Wiechman. Embedded software development is a multidimensional effort. Texas Instruments Incorporated, white paper, December 2011.
- [2] Texas Instruments. Digital Signal Processing Software and Development Tools Selection Guide. Texas Instruments Incorporated, March 2010.
- [3] [http://processors.wiki.ti.com/index.php/Main\\_Page](http://processors.wiki.ti.com/index.php/Main_Page).

## 第2章 DSP 可重用实时软件技术

第1章从宏观上讨论了复杂 DSP 系统开发需要考虑的各方面，其中的关键问题之一是可重用和实时软件设计。本章从算法标准和参考编程框架等方面讨论 DSP 可重用实时软件技术。

### 2.1 XDAIS 算法标准

#### 2.1.1 算法标准简介

数字信号处理器采用了哈佛结构和桶型移位器等专用的硬件逻辑，使其在语音、视频、通信等多个领域得到了快速的发展。为了实现实时复杂信号的处理算法，常采用直接在程序中嵌入汇编语句访问硬件外设。随着技术的发展，DSP 软件越来越复杂。传统的开发方法缺乏一致的标准，开发人员要将大量时间花在重复开发和集成工作上。虽然很多算法是多年科研成果，但是如果缺乏统一标准，当算法从一个系统移植到另外一个系统时，通常要做很大改动，算法的继承性和重用性都很差，不同产品开发商的算法运行环境相互不兼容，严重限制了 DSP 的发展和推广。

针对这种状况，算法在编码上必须遵循一定的规范。在处理器资源的使用上，也必须作些限制，这些限制和规范就是算法标准。符合标准的一个算法可以应用于不同的软硬件环境，在不修改算法编码下，可以重新连接和配置，并且独立于 I/O 设备。算法的性能以存储器和 MIPS 消耗来评估。一个算法要应用于多种操作系统，必须具备以下基本特征。

- 算法对于应用程序透明，相同的算法可以用于任何应用程序。
- 算法既能用于静态系统，又能用于动态系统。
- 算法能用二进制形式分发，即能以目标文件发行。
- 算法既能用于单任务，也能用于多任务，即算法具有可重入性。
- 算法的使用不需要重新编译，但可以重新连接和配置。
- 算法是 C 语言可调用的。
- 算法独立于任何特殊的 I/O 设备。

TI 公司的 eXpressDSP 实时软件技术中包括一套编写和使用 DSP 算法的标准 XDAIS，大大改善了算法移植和开发的效率，提高了算法重用性和通用性。eXpressDSP 中各组成部分与算法的关系如图 2.1 所示。DSP/BIOS 直接管理硬件接口，DSP 算法通过参考编程框架与应用程序打交道。DSP 算法标准 XDAIS 用于隔离框架和算法。从根本上讲，算法标准专注于将 DSP 资源的管理从算法本身抽象出来。这些资源包括：存储器的使用和分配、DMA 通道的使用等输入输出控制，以及关键控制寄存器的使用等。另外，如果希望算法能够在运行不同操作系统的环境中使用，算法就不能直接调用底层的操作系统服务。但是，XDAIS 标准



允许有限使用 DSP/BIOS，可以调用实时分析模块来动态调试算法，但不能调用线程或任务模块。

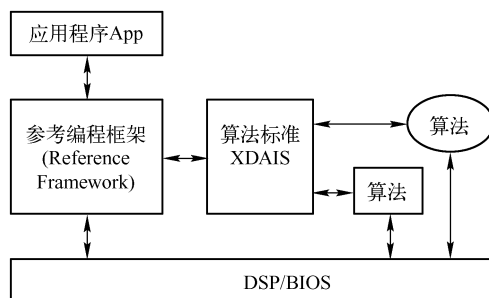


图 2.1 eXpressDSP 各部分与算法的关系

TI 定义的算法标准 XDAIS，通过一系列编程准则和建议，将算法和系统框架的接口标准化，对算法的开发者和使用者都大有益处。一方面大大减轻了算法使用者的系统集成工作，从而使产品开发周期大大缩短；另一方面，使算法提供商可以将自己的算法实现很好地封装起来，从而在保护自己知识产权的同时，又可以获得更广泛的客户。XDAIS 的主要优点有：与平台独立，从而使算法有很好的可移植性；方便算法的性能评估与比较；将存储器与执行环境等资源管理抽象出来，提高了算法的灵活性；统一的抽象接口，使算法易于集成到应用程序中；统一的命名规则，增强了代码的可读性，从而使不同人员开发的算法具有一致性；屏蔽了资源竞争，使多个算法协同工作更容易。

### 2.1.2 XDAIS 算法标准规则

XDAIS 算法标准的重点是设计出一套能够应用于所有算法的规则，并定义所有算法必须符合的资源管理接口。此外，还为 TI 的每个系列的 DSP 提供了一些具体的规则。

XDAIS 算法标准包括一系列准则和建议，分为以下五部分（具体准则和建议请参考《TMS320 DSP Algorithm Standard Rules and Guidelines》）。

(1) 通用编程规范，包括准则 1~6 和建议 1~2。该部分规范适用于所有 DSP 上的任何算法与应用程序。

(2) 算法模块规范，包括准则 7~18 和建议 3~5。为了将不同的算法在不修改源代码的前提下，集成到同一个应用中，该部分规范要求充分采用面向对象和基于组件的编程思想，包括命名约定、算法初始化方式和数据存储器管理机制。该部分规范认为，XDAIS 应用的基础组件是模块，XDAIS 算法必须实现 IALG 接口，框架通过 IALG 接口来了解并分配算法对存储器的需求。该部分规范适用于所有 XDAIS 算法。

(3) 算法性能描述规范，包括准则 19~24 和建议 6~10。XDAIS 兼容的算法只考虑 MIPS 和存储器消耗。所有 I/O、外设控制、设备管理和调度都由应用程序完成。因此，XDAIS 算法要描述数据空间的需求和最坏情况下的执行时间。另外，如果 CPU 处于中断禁止状态，调度不可能得到 CPU 控制权。所以，XDAIS 算法也需要尽量减少中断禁止时间，并告知最坏情形。该部分规范同样适用于所有 XDAIS 算法。

(4) 特定 DSP 规范，包括准则 25~34 和建议 11~15。由于算法经常用汇编来完成，以

便充分发挥 DSP 的性能，这往往使算法对不同的 DSP 有特殊的要求，这部分规范则就是针对这种情况而设定的。

(5) DMA 资源使用规范，包括 IDMA 准则 1~5 和 IDMA 建议 1。这部分规范针对准则 6 关于 DMA 的应用。由于 XDAIS 算法被设计为单纯的数据转换器，没有权力访问和使用片上的外设，包括 DMA，而是由参考编程框架来控制 DMA 的运行。为了让算法可以间接使用 DMA 资源，XDAIS 将算法对 DMA 的管理需求，抽象成一个接口 IDMA，类似于 IALG。框架通过调用 IDMA 接口函数，来了解算法对 DMA 资源的需求；另一方面，算法如果要知道 DMA 的运行情况，也可以调用框架实现的 ACPY。

上述准则和建议分成了三个层次，如图 2.2 所示。第一层为通用编程规范，包括 C 可调用、可重入以及不用绝对地址等；第二层为算法部件模型，包括模块、接口以及封装等规范；第三层为与特定 DSP 相关的算法资源使用规范。

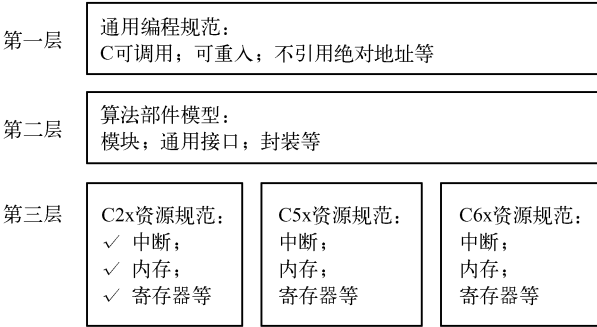


图 2.2 XDAIS 算法标准的层次

TI 的 CCS 提供了向导，帮助开发人员创建和使用符合 XDAIS 算法标准的 DSP 算法，包括以下七个步骤：第一步，要求开发者确定算法现在的状态，给模块起名，回答一些基本问题；第二、三、四、五步分别从算法用户、接口定义者和算法提供商的角度引导开发者实现算法的细节；第六步，描述怎样创建可交付使用的算法标准库；第七步，描述怎样表现算法的特点。具体过程请参考相关手册。

2.1.3 创建符合标准的 DSP 算法

为了创建符合 XDAIS 标准的算法，必须要为算法实现 IALG 接口。如果算法要使用 DMA 资源，还要实现 IDMA2 或 IDMA3 接口。为了能够进行现场调试和诊断，算法还要实现跟踪接口 IRTC。

应用程序必须通过 ALG 标准函数与算法通信，具体包含八个标准函数：ALG\_activate()、ALG\_deactivate()、ALG\_create()、ALG\_delete()、ALG\_control()、ALG\_setup()、ALG\_init()、ALG\_exit()。这八个函数仅仅定义了函数名或指针名，实际函数内容是通过相对应的具体算法的 IALG 接口函数实现的。其中函数 ALG\_activate() 和 ALG\_deactivate() 是为了解决多个通道的算法重入问题，通过这两个函数在调用前后对参数做保存和恢复，以及在动态和静态内存间进行数据的拷贝和搬移。ALG\_activate() 函数激活实体对象，主要是初始化动态内存中的数据，以供算法使用；ALG\_deactivate() 函数撤销实体对象，将下次执行算法时需要的数据从动态内存保存到静态内存中。ALG\_control() 函数则通过命令控制字对参数进行动态

调整或给出算法状态等信息。

IALG 是 XDAIS 定义的抽象接口，符合标准的算法必须要实现这一接口。IALG 接口允许算法定义自己的存储器资源需求，为应用程序提供了创建和管理算法实例的方法，使算法可以运行于抢占或非抢占、静态或动态系统等各种环境中。IALG 接口最重要的是通过定义 IALG\_Fxns 结构体，封装了一个函数指针表 v-table（图 2.3），定义了满足算法标准的接口函数指针，以供上述 ALG 的八个标准函数或者应用程序通过这个接口调用算法。

```
typedef struct IALG_Fxns{
    Void *implementationId;
    Void (*algActivate) (IALG_Handle);
    Int (*algAlloc) (const IALG_Params*, struct IALG_Fxns**, IALG_MemRec*);
    Int (*algControl) (IALG_Handle, IALG_Cmd, IALG_Status*);
    Void (*algDeactivate) (IALG_Handle);
    Int (*algFree) (IALG_Handle, IALG_MemRec*);
    Int (*algInit) (IALG_Handle, const IALG_MemRec*, IALG_Handle,
        const IALG_Params*);
    Int (*algMoved) (IALG_Handle, const IALG_MemRec*, IALG_Handle,
        const IALG_Params*);
    Int (*algNumAlloc) (Void);
} IALG_Fxns;
```

图 2.3 IALG\_Fxns 结构定义

算法实现 IALG 接口时，必须声明和初始化 IALG\_Fxns 结构体类型的全局变量 XYZ\_IALG（其中 XYZ 是代表算法模块的前缀）。全局变量 XYZ\_IALG 中，包括算法实例标识符和函数指针表 v-table，算法使用者可以通过这个函数指针表来管理具体的算法实例。算法开发人员要根据 v-table 的格式实现自己的接口函数。v-table 的函数指针与上述 ALG 的八个标准函数相对应，其中有些函数是可选的，但 algAlloc()、algInit() 和 algFree() 三个函数是算法模块必须实现的。IALG 接口函数的调用顺序如图 2.4 所示。首先创建算法实例，应用程序应传递参数给 algAlloc() 和 algInit() 函数，不同算法的参数不同，一般包括与算法相关的主要参数，必须根据 IALG\_Params 结构体类型来定义。例如，对 G723 语音编解码算法而言，参数有每帧长度和编码系数帧长度等，其参数结构体定义如下：

```
typedef struct Ixx_Params {
    int          size;                /* Size of this structure */
    XDAS_UInt16 frameLen;             /* Length of frame buffer */
    XDAS_UInt16 codeLen;             /* Length of code parameter buffer */
} Ixx_Params;
```

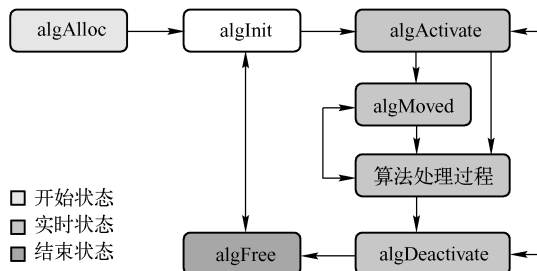


图 2.4 IALG 接口函数的调用顺序



一旦算法实例被创建好,就可以用于实时数据处理。如果存在其他算法处理函数入口,就要通过调用 `algActivate()` 函数,激活实例对象。处理函数执行完后,应用程序可以调用 `algDeactivate()` 函数,来重用算法实例对象的动态存储器。另外, `algControl()` 提供了算法实例对象接收实时状态信息的方法, `algMoved()` 使应用程序能把算法实例对象移动到不同的物理存储器中。最后调用 `algFree()` 函数释放资源。

### 1. `algAlloc()`

算法必须实现 `algAlloc()` 函数,声明自己的存储器资源需求。函数原型为:

```
int algAlloc(const IALG_Params *, IALG_Fxns **, IALG_MemRec);
```

其中,第一个参数与特定的算法有关,表示创建算法实例时的参数表指针,这个指针可以为空。第二个参数用于存放可选的输出参数, `algAlloc()` 可以用它把 IALG 父函数指针返回给用户程序。第三个参数为算法所需的全部缓冲区的存储器记录表,每个记录包括大小、对齐方式、类型和存储空间。当函数调用成功时,返回初始化的存储器记录数(应大于或等于1),否则表示调用失败。在下面的例子里,申请一块静态存储空间,这块存储空间必须足够大,以便满足在执行算法实例时,需要定义的对象和工作缓冲区的需求。`algAlloc()` 函数通过填 `memTab` 表来告知应用程序,算法所需要的存储空间。根据此信息,应用程序可以在调用函数初始化对象之前,分配所需的存储空间。所有对象的访问,都通过一个指向对象的指针来进行,这是编写可重入代码的好方法。

```
int G723ENC_TI_algAlloc (const IALG_Params * algParams,
                        IALG_Fxns ** pf, IALG_MemRec memTab[])
{
    /* Request memory for G723ENC instance object */
    memTab[0].size = sizeof(G723ENC_TI_Obj);
    memTab[0].alignment = 0;
    memTab[0].space = IALG_DARAM0;
    memTab[0].attrs = IALG_PERSIST;
    return(1) /* return number of memory blocks requested */
}
```

再来看 `G723ENC_TI_Obj` 的定义:结构体的第一个域 `IALG_Obj`,是一个指向 `v-table` 的指针,注意这个参数必须是第一个域。当创建一个算法实例时,由应用程序来初始化这一指针,使其指向 `v-table`。同样, `G723ENC_TI_Obj` 也必须占用第一个存储块 `memTab[0]`。

```
typedef struct G723ENC_TI_Obj
{
    IALG_Obj ialg;           /* Points to the v-table */
    IG723_Rate workingRate;  /* 5.3 or 6.3 kbps */
    XDAS_Bool hPFilter;      /* High Pass filter on/off */
    XDAS_Bool VAD;           /* Voice activity detection on/off */
    .....                  /* specifics to the implementation */
} G723ENC_TI_Obj;
```

## 2. algInit()

algInit() 函数完成创建一个算法实例所需的初始化。如果函数返回成功，算法实例对象就已经准备好进行数据处理了。函数原型为：

```
int algInit(IALG_Handle, IALG_MemRec, IALG_Handle, IALG_Params *);
```

其中，第一个参数为算法实例对象的句柄，应用程序调用了 algInit() 之后，可以通过此句柄使用算法实例对象。第二个参数与 algAlloc() 函数中相应参数的含义相同。第三个参数表示算法实例的父对象的句柄，当算法实例没有父对象时，此参数为空 (NULL)。最后一个参数为与特定算法有关的参数表指针，含义与 algAlloc() 函数中的相应参数相同，但 algInit() 函数必须假定默认创建参数。参考下面的例子：

```
int G723ENC_TI_algInit(IALG_Handle handle, const IALG_MemRec
    memTab[], IALG_Handle p, const IALG_Params *algParams)
{
    G723ENC_TI_Obj *enc = (void *)handle;
    const IG723ENC_Params *params = (void *)algParams;
    if (params == NULL)
    {
        params = &IG723ENC_PARAMS; /* set default parameters */
    }
    /* Copy creation params into the object */
    enc->workingRate = params->rate;
    enc->hpfFilter = params->hpfEnable;
    enc->VAD = params->vadEnable;
    g723EncInit(enc); /* Initialize all other instance variables */
    return(IALG_EOK);
}
```

可见，句柄就是 G723ENC\_TI\_Obj 对象的指针。另外，如果 algAlloc() 返回的父函数指针不为空，第三个参数应填入有效的父对象的句柄。而且，algInit() 函数不能被同一算法实例的其他函数抢占。

## 3. algFree()

algFree() 函数释放 algAlloc() 分配的存储块。函数原型为：

```
int algFree(IALG_Handle, IALG_MemRec *);
```

由算法来设置每个存储块的地址和大小，从而保证在删除对象实例时，不产生存储空间泄漏。同样，algFree() 函数不能被同一算法实例的其他函数抢占。参考下面的例子：

```
int G723ENC_TI_algFree(IALG_Handle handle, IALG_MemRec memTab[])
{
    G723ENC_TI_Obj *enc = (Void *)handle;
    algAlloc(NULL, NULL, memTab); /* Fill the memTab struct */
}
```

```

    memTab[0].base = (Void *) &enc;
    return(1);
}

```

#### 4. 模块特定接口

除了 IALG 接口，不同算法模块还要实现自己特定的服务提供者接口 (SPI)。一个算法模块可以实现一个或多个接口，不同接口对应不同的头文件。例如，TI 为 ITU G. 723.1 提供了 SPI 示例：IG723ENC。为了用一个完整的算法实现 IG723ENC 接口，还需要创建 control() 和 encode() 函数，参考下面 encode() 的例子：

```

XDAS_Bool G723ENC_TI_encode(IG723ENC_Handle handle,
XDAS_UInt16 *in, XDAS_UInt16 *out)
{
    G723ENC_TI_Obj *enc = (void *) handle;
    if(encoder(enc, in, out) /* do the processing */)
        return(XDAS_TRUE);
    return(XDAS_FALSE);
}

```

下一步要定义和初始化 IG723ENC 的函数指针表 v-table。由于 G723ENC\_Fxns 是在 IALG\_Fxns 的基础上扩展的，IG723ENC 的 v-table 必须要包括 IALG 的 v-table 中的函数。

```

#define IALGFXNS \
&G723ENC_TI_IALG,          /* module ID */\
NULL,                      /* activate */\
    G723ENC_TI_algAlloc,    /* alloc */\
NULL,                      /* control */\
NULL,                      /* deactivate */\
G723ENC_TI_algFree,        /* free */\
G723ENC_TI_algInit,        /* init */\
NULL,                      /* moved */\
NULL                       /* numAlloc */\
IG723ENC_Fxns G723ENC_TI_IG723ENC =
{
    IALGFXNS,               /* IALG functions */\
    G723ENC_TI_control,
    g723ENC_TI_encode
} G723ENC_TI_IG723ENC;
asm(“_G723ENC_TI_IALG.set _G723ENC_TI_IG723ENC”);

```

v-table 的第一个域是表的地址，它是执行的唯一标识符。在 v-table 中只定义了三个必须的 IALG 函数和特定模块函数，所有其他的函数指针都被设置为空指针。asm 声明中，定义符号 G723ENC\_TI\_IALG 等同于 G723ENC\_TI\_IG723ENC，意味着 G723ENC\_TI\_IG723ENC

和 G723ENC\_TI\_IALG 共享同一个 v-table。如果要调用 v-table 中的 IALG\_Handle 函数，只需访问 IALG\_Fxns；如果要调用 encode() 和 control() 两个函数，需要访问 IG723ENC\_Handle。

## 5. 内存分配方式

XDAIS 标准将算法分为动态和静态两种内存分配方式，以满足一个应用同时调用不同的算法库以及快速的算法转换和数据传递，也可以减少内存的浪费，提高内存的重复使用率。

静态方式是封装成标准算法库时定义的，包括的主要功能为：分配算法需要的内存块并定义一个内存表，确定每块内存的位置、名称、大小和属性（动态或者静态）；设置内存表内定义的各内存块物理首地址为算法模块定义的数组的首地址，完成算法模块的内存地址分配；将内存表的首地址赋值给算法实例句柄指针，以指针操作代替实际物理地址操作。静态分配仅仅定义了在建算法实例时用到的所有内存块与实际内存的映射关系，并没有实际在内存空间中进行真正的分配动作，重点在以指针操作代替实际物理地址操作，将内存的分配与算法对象联系起来，避免了在算法实例中出现实际物理地址，导致算法的不可移植，同时也为不同算法共同使用内存和合理分配内存提供了简单直观的安排方式。

动态方式是在调用算法模块、创建算法实例对象时进行的，包括的主要功能为：动态创建一个实例对象，将算法模块中的 v-table 和参数结构体传递给该实例对象；为算法实例提供固定和共享的两种不同数据存放方式的内存块；并调用 IALG 接口的各种函数操作内存块内的数据，以完成算法输入数据传递和结果的保存等功能。动态分配流程如图 2.5 所示。首先需要使用 DSP/BIOS 配置工具配置好动态内存定位时的内部和外部存储堆栈，以便在参考框架执行算法初始化时一次性设置好。然后确定算法需要的最大内存块数目；并为内存描述表定位，即确定每块内存的大小、读取方式（单向读写、双向读写）、属性（动态、静态、一次性写入）、首地址等；接着初始化新创建的对象：设置对象句柄指向内存相应的首地址、设置对象函数结构体为默认 v-table，完成 v-table 的正确传递，使算法实体可以调用算法标准库中的各种函数、调用 algInit() 函数初始化算法实例，完成算法实例对象的创建过程。algActivate() 和 algDeactivate() 函数具体管理动态和分享的内存，activate() 函数准备好动态数据供算法运行使用；deactivate() 函数保存动态数据中下次还会被用到的数据。动态内存分配的重点在于对动态内存需求的定义以及从算法模块传递 v-table 和参数结构体给算法实例。

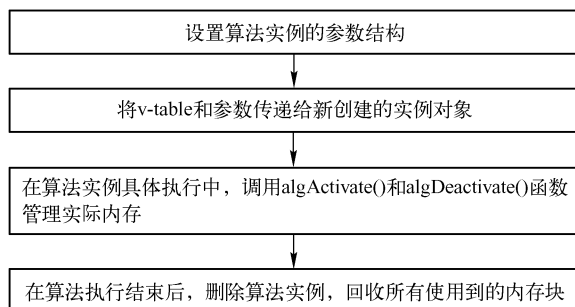


图 2.5 动态分配流程

### 2.1.4 XDAIS 算法实例

下面以 G722 编码为例, 介绍如何编写一个 XDAIS 算法。

**第一步, 定义参数结构体**, 主要包括编码算法中数据输入和输出缓冲区的长度, 以及编码速率。因为不同的编码速率直接影响编码的结果数据大小, 所以在参数结构中预先设置了编码速率。结构体定义如下。

```
typedef struct IG722ENC_Params {
    Int          size;           /* Size of this structure */
    XDAS_UInt16  frameLen;       /* Length of input buffer */
    XDAS_UInt16  codeLen;        /* Length of code result buffer */
    IG722_Rate  rate;           /* Working rate */
} IG722ENC_Params;
```

**第二步, 定义算法核心函数**G722ENC\_YX\_encode, 作为应用层调用算法的接口, 具体流程如图 2.6 所示。函数原型为: Void G722ENC\_YX\_encode ( IG722ENC\_Handle handle, int in[ ], int out[ ] );

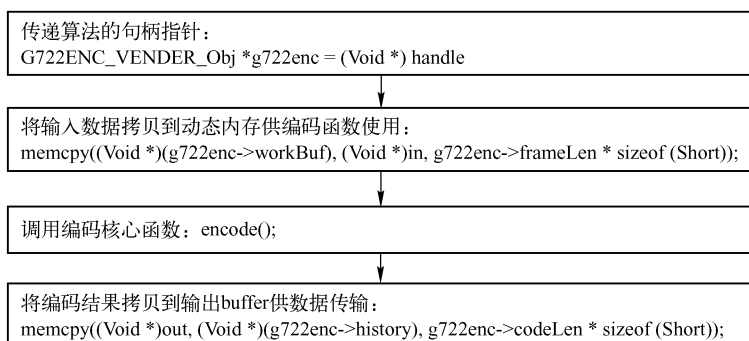


图 2.6 G722ENC\_YX\_encode 流程

**第三步, 定义 ALG 接口函数。**

1. G722ENC\_apply: 编码执行函数接口, 封装了应用程序调用 G722ENC\_YX\_encode 的接口, 原型定义如下。

```
extern XDAS_Int16 G722ENC_apply( G722ENC_Handle handle, XDAS_Int16 * in, XDAS_Int16 * out );
```

2. G722ENC\_control: 编码控制函数接口, 参数为编码算法的对象, cmd 命令参数格式, 状态参数格式, 原型定义如下:

```
extern XDAS_Bool G722ENC_control( G722ENC_Handle handle, G722ENC_Cmd cmd, G722ENC_Status * status )。
```

3. G722ENC\_create: 编码实例对象创建函数接口, 传递了编码算法对象的 v-table 结构体、参数结构体格式、调用该函数, 将通过 ALGRF\_create 标准函数, 创建基于实参的标准

算法实例对象，定义如下。

```
static inline G722ENC_Handle G722ENC_create( const IG722ENC_Fxns * fxns, const G722ENC_Pa-
rams * prms)
{ return ( ( G722ENC_Handle ) ALGRF_create( ( IALG_Fxns * ) fxns, NULL, ( IALG_Params * )
prms) ); }
```

4. G722ENC\_delete: 编码对象删除函数接口，传递了对象的句柄指针，调用该函数，将通过 ALGRF\_delete 标准函数，删除该对象，定义如下。

```
static inline Void G722ENC_delete( G722ENC_Handle handle)
{ ALGRF_delete( ( ALGRF_Handle ) handle ); }
```

#### 第四步，实现 IALG 接口。

1. 对 IALG\_Fxns 的 v-table 进行扩展，定义头文件中的算法自身 v-table，包含一个标准的 IALG\_Fxns 结构体，以及 G722 编码算法的核心函数 encode。

```
typedef struct IG722ENC_Fxns {
    IALG_Fxns   ialg;           /* IG722ENC extends IALG */
    Void * encode();
} IG722ENC_Fxns;
```

2. 创建算法实例对象和句柄指针，以实现算法的控制和操作。算法对象定义如下。

```
typedef struct IG722ENC_Obj {
    struct IG722ENC_Fxns * fxns;
} IG722ENC_Obj;
```

再定义一个结构体指针 IG722ENC\_Handle 作为算法对象句柄。

```
typedef struct IG722ENC_Obj * IG722ENC_Handle;
```

3. 在源文件中，初始化扩展的 v-table。

```
IG722ENC_Fxns G722ENC_VENDER_IG722ENC = {
    IALGFXNS,
    G722ENC_VENDER_encode,
};
```

再用宏定义实现算法实例对标准 v-table 的扩展，并具体化每个函数的实际功能。

```
#define IALGFXNS
    &G722ENC_VENDER_IALG,           /* module ID */
    G722ENC_VENDER_activate,        /* activate */
    G722ENC_VENDER_alloc,           /* algAlloc */
    G722ENC_VENDER_control,         /* control ( NULL = > no control ops ) */
    G722ENC_VENDER_deactivate,      /* deactivate */
```

```
G722ENC_VENDER_free,           /* free */
G722ENC_VENDER_initObj,        /* init */
G722ENC_VENDER_numAlloc        /* numAlloc */
```

当我们调用 G722 编码算法模块时，调用算法执行函数 G722ENC\_apply()，函数体内调用 ALG\_activate()，代码如下。

```
Void ALG_activate( ALG_Handle alg)
{
    if ( alg -> fxns -> algActivate != NULL)
    {
        alg -> fxns -> algActivate (alg);
    }
}
```

当运行 alg -> fxns -> algActivate (alg) 语句时，就会调用到用户扩展的 v-table 中的宏 IALGFXNS: G722ENC\_VENDER\_activate 函数，完成激活编码算法实例对象的内存初始化工作。也就是说应用程序调用 ALG 接口的标准函数时，最终调用的是算法相对应的 IALG 接口的具体代码函数。

4. 实现各个 IALG 函数的具体功能。

以算法实例激活函数为例：Void G722ENC\_VENDER\_activate (IALG\_Handle handle); 此函数初始化动态内存中编码函数的输入数据，将应用层从 PIP 层得到的输入数据拷贝到动态内存，供 G722ENC\_VENDER 算法实体使用，具体功能如图 2.7 所示。其他函数的实现细节不再赘述。

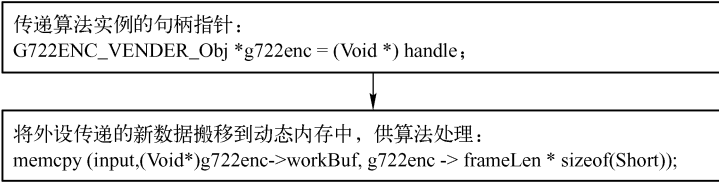


图 2.7 算法实例激活函数流程

另外，注意实现动静态内存分配。

G722 编码算法定义了 3 个内存块，组成了一个内存表 memTab[3]：memTab[OBJECT]就是 memTab[0]；memTab[HISTORY1]就是 memTab[1]；memTab[WORKBUF1]就是 memTab[2]。

```
#define    OBJECT      0
#define    HISTORY1    1
#define    WORKBUF1    2
#define    NUMBUFS     3
```

G722ENC\_VENDER\_alloc 函数描述了 G722ENC\_VENDER\_Obj 对象需要的内存结构，并对每块内存的长度、存储属性做了详细的定义，将 G722ENC 对象指针句柄需要的内存、编码算法存储处理结果需要的内存和编码算法存储输入数据需要的内存做了详细的定义。

其中编解码的工作缓冲区都定义为动态内存块，用于存储编解码函数的处理对象数据，



因为这些输入数据是临时的，当不调用编码算法时，这些数据占用的空间可以被其他算法占用，提高算法对内存的重复利用率。history buffer 都定义为静态内存块，用于处理编解码的结果数据，因为这些编解码的结果可能是其他算法的输入数据或者等待传输给外设，这种等待的时间可能是不定的，所以需要固定的内存块来保存。整个 G722 算法编解码对动态和静态内存使用的数据流程如图 2.8 所示。

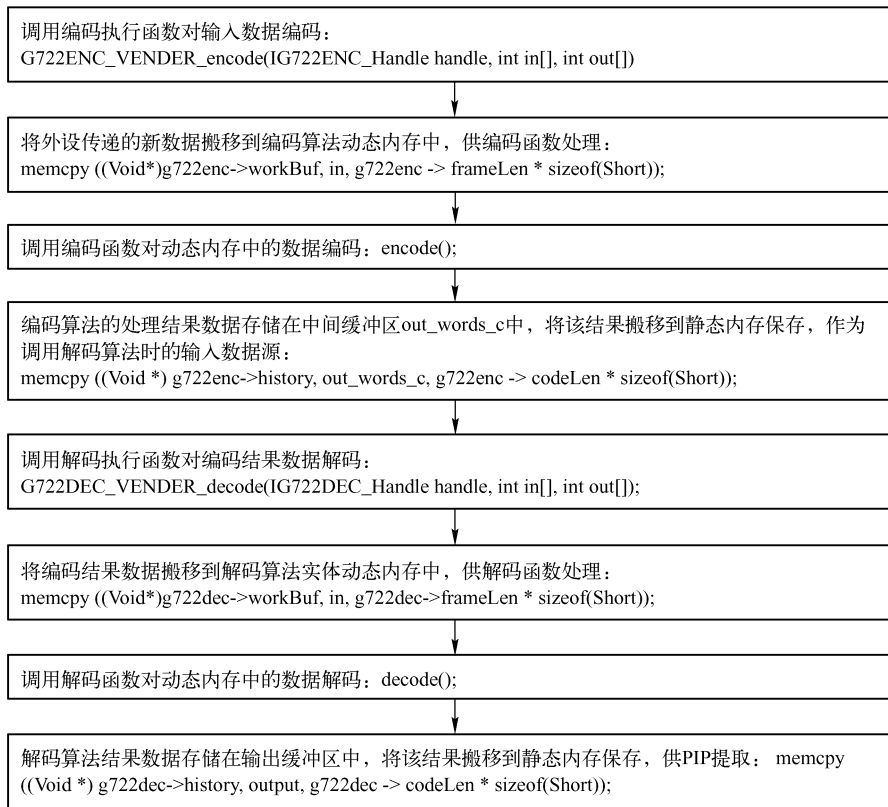


图 2.8 G722 算法内存使用数据流程

## 2.2 参考编程框架

TI 公司的 eXpressDSP 实时软件中的参考编程框架（RF，Reference Framework）定义为从属于一个典型的 DSP 软件系统的底层基础软件，它将应用程序以及符合标准的 DSP 算法连接起来，管理系统资源，进行硬件抽象，处理输入输出并且为算法提供容器。RF 包括设计好了的、可重用的 TMS320C5000 和 TMS320C6000 等数字信号处理器的 C 语言源程序代码，可以作为使用 DSP/BIOS 和 TMS320DSP 算法标准（XDAIS）的入门套件。开发者首先选择最适合自己系统需要的参考框架，然后在这个框架基础上开发自己的目标程序，并将符合 XDAIS 标准的算法移植到 DSP 目标板上。通用的组件如设备驱动、内存管理器和信道封装等都已经在这个框架中预先配置好了，开发者可以只把注意力放在系统特殊的要求上，确认芯片和系统的特性是否适合目标程序，从而提高总的开发效率。



### 2.2.1 RF 简介

1.2 节介绍了 TI 公司的 eXpressDSP 实时软件和开发工具,对加速 DSP 软件开发提供了机遇。尽管 DSP 的应用已远远超过传统的电话和通信领域,但是只需要有限的几个基础架构就足以构建大部分应用系统。正是因为认识到这一点, TI 为 eXpressDSP 定义了参考编程框架,以便将底层基础组件连接到一起。开发者只需选择正确的参考编程框架,就可以适合不同应用的需要。

#### 1. 建一个村舍还是城堡?

我们用建筑物来类比解释定义不同参考编程框架背后的逻辑。当修建一个村舍时,木头和砖这样的简单材料就够了。设计准则的要点是小的物理空间、有限数量的房间,以及低费用。但是如果修建一座城堡呢?建筑材料和设计标准就大大的不同了。建筑师对这两种建筑使用完全不同的设计方法:什么时候你见过在村舍附近有护城河和吊桥呢?然而,这些建筑工程依然有相同的特性:例如都需要使用电源和水管,而且都要满足工业标准的建筑材料。

基于低价格的 TI DSP 如 TMS320C5401 的 DSP 工程就像村舍。这个很便宜的 DSP 器件每秒能执行 50 百万条指令以及 8K 内存。然而,设计者面临挑战——如何在如此局限的环境中构建 eXpressDSP 的基本组件?比如: DSP/BIOS、XDAIS 和 CSL? 开发者只能根据程序嵌入深度、需要运行的算法数目和要处理多少信道的数据进行折中。

而另一端,新一代 TMS320C64x+ 提供了数千 MIPS 的运算能力以及很大的内部和外部存储器,就像城堡。设计者需要更大的灵活性以便能创造性地发挥 DSP 的高性能,算法的数量可能很大并且在运行时有明显变化,处理信道的数目也会很多。虽然此类工程也会用到 DSP/BIOS、XDAIS 和 CSL,但是可能具体的方式与村舍项目大大不同。

就像村舍和城堡都需要电源和水管一样, DSP 应用可能都使用 DSP/BIOS、XDAIS 和 CSL。但是,类似城堡比村舍需要更多复杂的电力和供水系统,一些 DSP 应用软件也可能比另一些需要具有更多的弹性和性能。

#### 2. DSP 软件体系特性

对大多数 DSP 系统而言,只需要关注 6 个关键特性就可以决定其软件架构。例如,两通道语音编码系统的软件架构几乎和两通道 MP3 解码器的软件架构相同,也与两通道回声消除系统的软件架构一样。值得注意的是,此处讨论的软件架构不涉及面向用户的终端应用。下面讨论这 6 个方面。

1) **系统中会使用多少种算法?** 如果只使用一到两种算法,用一些基本技术就能最小化存储器空间需求。另一方面,如果使用许多种算法或者可变的算法集,则需要更复杂的技术来管理这些算法。

2) **系统中要操作多少通道/信道运行?** 同样地,如果只有一到两个通道/信道,可以进行确定的假设和优化。而对于大数目或更灵活的通道/信道,需要更先进的方法和手段。

3) **系统中需要动态创建对象和资源分配,还是静态配置就可以满足要求?** 动态对象创建意味着在运行时创建对象,运行后删除。这种机制会影响存储器的分配:动态系统通常使用 DSP/BIOS 的 MEM\_alloc() 函数动态分配存储器。然而,如果可以预先完全指定使用的对象和函数,系统就能静态配置。静态配置可以显著的减少系统存储空间,例如一个函数不用

像动态系统那样创建多个运行时的副本，从而节约了珍贵的片上程序存储器。

4) **系统运行于单一速率还是多重速率?** 考虑一个系统所需要的帧率非常重要。在一个简单的系统里，可能所有算法的运行帧率都是 10ms，这种系统可以进行确定的优化。然而，一个多速率系统可能有一个函数运行于 10ms，而另一个运行于 16.6ms。这种系统更复杂，虽然 DSP/BIOS 可以很好地支持这类系统，但需要加载额外的 DSP/BIOS 模块才能保证系统的正确工作。

5) **系统的内存有多大?** 有些 DSP 只有 8K 或者 16K 的内部存储器，而大量新设备有数百 KB 的内部存储器。因此，在功能和存储空间之间进行折中总是很重要的。如果空间紧张，如一个 8K 系统，首先应满足储存空间的要求，故必须在功能方面作出让步。然而，一个有 256K 内部存储器的系统，就应更多地考虑系统的功能和灵活性。

6) **系统是否需要外部控制?** 如果需要外部控制，是用一个简单的线程操作通信机制，还是需要进行全部特性的连接控制？例如，有的系统包含一个通用处理器（GPP），如 PowerPC 或 MIPS。在这种情况下，DSP 和 GPP 如何通信就非常重要，RF 应该提供进行通信的工具而不是让开发者从头创造通信协议。另一些系统则使用简单的低优先级的控制线程来响应外部控制，允许外部处理器（如主机 PC）控制 DSP 运行。

回答这些问题就可以知道要建立的 DSP 系统的类型。这些答案虽然有多种可能的组合，但只有一些组合常用。eXpressDSP 的参考编程框架就是对 DSP 软件进行基础配置，提供给开发者这些常用组合以简化开发。

### 3. RF 的通用特性

在讨论不同参考编程框架的细节前，先说明所有 RF 都具有的通用特性。

- 100% C 代码：参考编程框架是完全用 C 语言写的，因此易于修改，能够轻松地选择满足自己应用系统需要的模块。RF 容易移植到不同的目标平台和 TI 下一代 DSP，这是使用了 DSP/BIOS 对硬件进行抽象而带来的好处。
- 完整解决方案：每个参考编程框架均提供完整解决方案。注意 RF 的解决方案包括简单的流行算法如 FIR 滤波器，此处的策略是说明很容易将这些算法用更复杂的 XDAIS 算法替代。
- 易于修改：开发者希望知道怎么样能快速生成面向特定应用的通用例子程序，而不需要理解系统的底层细节。这一点可以通过重点关注以下步骤实现：转换算法、添加信道、修改驱动等。
- 充分的文档：使用者关注的是处理过程的 MIPS 和存储空间需求等细节。这些细节体现为在所有框架中提供一致和正确的系统代价描述。

### 4. RF 的等级

根据前述 DSP 系统的 6 个关键特性可以将系统分类，构建不同架构的 DSP 软件系统需要 5~10 个等级的参考编程框架，TI 定义的第一、三、五等级的 RF 特性如表 2.1 所示。

表 2.1 不同 RF 等级的特性

设计参数	RF1	RF3	RF5
静态配置	√	√	√
动态创建对象	×	×	×
静态存储管理	√	√	√

续表

设计参数	RF1	RF3	RF5
动态存储管理	×	√	√
通道/信道数	1 ~ 3	1 ~ 10	1 ~ 100
算法数目	1 ~ 3	1 ~ 10	1 ~ 100
最小内存需求	√	×	×
单一速率操作	√	√	√
多重速率操作	×	√	√
线程阻塞	×	×	√
实现控制	×	√	√

RF1（紧凑型框架）：这个框架对存储空间的需求最小，但仍然包括了所有 eXpressDSP 的要素。它设计为使用静态对象配置，不支持实时或者动态创建对象。内存管理也是完全静态的，不支持动态内存管理。此框架优化为支持比较少数目的信道和算法，丢弃了一些灵活性，换来的是最小的绝对存储空间。实际上，一个 RF1 的框架在 TMS320C54x 平台上最小能达到 3.5K 字。RF1 不支持线程抢占或阻塞；只支持单一速率，没有提供控制线程或者与 GPP 通信的机制。数字助听器或低价格的网络音频播放器等应用可以采用这个框架。

RF3（灵活型框架）：这个等级最大的变化是放松了绝对最小存储空间的要求，增加了灵活性。尽管 RF3 依然只支持静态对象创建，但已经加上了对动态存储器的管理。这样，可以实时配置数据缓冲器。RF3 允许使用更多的信道（典型的到 10）和更多不同的算法（同样也到 10）；支持多速率操作，也就是说不同的算法可以运行在不同的速率下，注意同一个信道中 RF3 只支持单速率。例如，一个算法运行于 10ms/帧，另一个运行于 20ms/帧。而且，RF3 提供了附加的线程以允许其他处理器（如主处理器）来控制 DSP。然而，灵活性的代价是加大了存储空间需求：典型 RF3 执行框架的大小在 TMS320C54x 平台上约为 11K 字。

RF5（扩展型框架）：这个框架设计为考虑最大的灵活性和扩展性，而不关注系统存储空间。与 RF3 相同，RF5 支持静态对象创建，以及静态和动态存储器管理。RF5 可以支持动态创建对象，但需要预先静态配置。RF5 的结构可以支持上百的信道和算法，也支持单速率和多速率操作。所有的线程可抢占和阻塞，支持控制线程。RF5 使用了超过 70% 的 DSP/BIOS 模块。显然，所有的灵活性增加是以系统的存储空间增加为代价的。然而，采用 DSP/BIOS 模块的静态配置可以让存储空间的需求足够小，从而使得基于 RF5 的应用系统可以运行于小型 TMS320CC6000 和 TMS320C55x 设备上。

## 5. RF 的组件

RF 属于 eXpressDSP 实时软件的组成部分，运行于 DSP/BIOS 之上，与设备或者驱动之间有清晰的隔离，通过标准驱动模式与设备接口，使得 XDAIS 算法的集成和转换更容易。

RF 的组件包括存储管理和复用，信道抽象以及算法管理。存储器管理和复用是 RF 的重要组件，这是一项在多数存储受限系统中的关键技术。当开发 DSP 软件时，简单地选择适当的 RF 开始，会使要处理的存储管理问题简化很多。

RF 为 IALG 接口提供了两种具体实现：ALGMIN 和 ALGRF。ALGMIN 适合 RF1，是 IALG 的一个最小实现版本。而其他 RF 通常采用 ALGRF 来创建、配置和删除 XDAIS 算法。

两者的比较见表 2.2。

表 2.2 RF 中两种 IALG 实现方案的比较

	ALGMIN	ALGRF
存储管理	静 态	动 态
关键特性	超级紧凑； ALGMIN_new 是关键初始化函数，它调用 algInit但不调用 algAlloc	使用 DSP/BIOS MEM 模块进行动态分配； 比 CCS 提供的算法接口的实现更小； 支持动态存储共享

RF 都需要某种类型的信道管理。如果事先能了解会使用多少信道，就会更容易地进行设计优化。例如，一个只处理 1~3 个信道的简单系统，信道调度可以用低负载的 DSP/BIOS 的 HWI 和 IDL 模块。对于较多信道数目的系统，使用 SWI 模块比较合适，虽然 SWI 需要更多的存储空间。对于信道动态改变的复杂系统，则使用 TSK 模块最适合。在每个参考编程框架中，这些设计选择可以帮助设计者以最适合的方式实现运行更快的系统。

算法管理器与信道管理器类似，管理若干数目的符合标准的 DSP 算法。遵循 XDAIS 标准的算法不能直接操作任何外围硬件，必须通过 RF 的资源管理接口 IALG（存储器管理）和 IDMA（DMA 资源管理）等才能运行。

### 2.2.2 RF1——紧凑型编程框架

RF1 的目的在于：在有严格存储器资源限制的 DSP 上实现非常紧凑的消费类产品，同时让设计者保留 eXpressDSP 软件技术的优点，以方便集成符合标准的 DSP 算法。

默认情况下，RF1 通过一个简单算法，如有限冲激响应（FIR）滤波器，来控制一个单一信道中的数据流。FIR 是 TI 提供的符合 eXpressDSP 的算法。通过配置和修改 RF1，如改变或增加算法、增加信道等，使 RF1 能满足具体应用的要求。

#### 2.2.2.1 RF1 概述

使用 RF1 的典型应用一般对成本敏感，常选用低价的 DSP 芯片，不用外部存储器。包括 RF1 的完整应用程序可以装入小于 8K 字的存储器中，这种存储容量，即使 TI 公司最便宜的 DSP 芯片 TMS320VC5401 也能满足。这时 DSP 没有可利用的外部存储器，所有的应用程序都必须装载到内部的存储器中。然而，最小的应用也包含一些基本的软件，如 DSP/BIOS、片级支持库（CSL）、XDAIS 算法以及连接它们的框架。

既然这些 DSP 应用资源有限，为什么开发者还要考虑 RF1？开发自定义程序架构和驱动，从而留下尽可能多空间供算法处理使用，岂不是更好？

在一些情况下，答案也许是肯定的。但是，典型的系统是随时间发展变化的。一个单通道的 MP3 解码器可能会发展成支持多通道标准，例如高级音频译码标准（AAC）或者 Windows 媒体格式标准（WMF）。一个数字助听器也许需要根据用户的听力参数进行调节，以适应用户的听力习惯。

当系统发展变化时，结构化编程的好处就越来越明显。开发者既可以在已有的框架上构建新程序，也可以轻松地将应用程序移植到更高级别的框架上。无论哪种情况，使用 RF1 的存储器空间占用情况都比自定义的程序架构要好一些。



RF1 有以下一些关键特征。

- 极小的绝对存储器空间。RF1 使用一系列专门的技术减小应用程序的存储器空间占用。目的在于支持使用小的存储器空间的低价格的 DSP 芯片。
- 支持 XDAIS 算法。虽然 RF1 的目的是最小化存储空间需求，仍然支持一定数量的符合 XDAIS 标准算法、RF1 静态配置算法需要的存储器。
- 基于 PIP 的 I/O。PIP 模块比 RF5 中的 SIO 模块更简单，但灵活性较少，然而 PIP 的开销要小一些。
- 调试容易。RF1 定义清楚的结构使用户能够方便地调试。此外，UTL 模块允许快速地转变调试级别。
- 容易修改 I/O 驱动。RF1 用的 IOM 模型允许用户将迷你驱动和 PIO 适配器相连接。

每种框架都可以适用于多种应用，适合 RF1 的应用系统如图 2.9 所示，包括数字助听器、MP3 播放器、智能玩具、无线婴儿监控器、数字扫描仪、单通道简单图像处理、低功耗付费电话、手持式患者监控系统、扩音器的回声消除等。

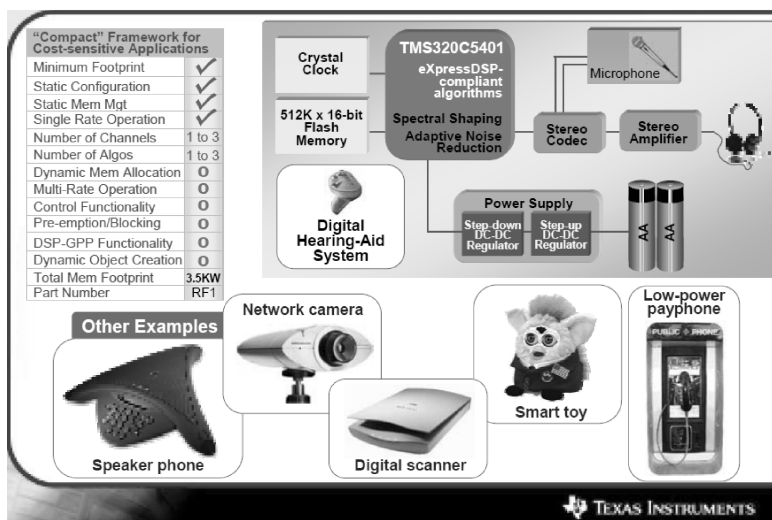


图 2.9 RF1 的应用系统

### 2.2.2.2 安装和运行 RF1

RF1 的应用适合运行在 C5402 DSK、C5416 DSK、C5510 DSK 上，也可以配置在其他 DSP 平台上。下面的步骤说明怎样将主机与 DSP 硬件连接。

- 关闭你的 PC。
- 将正确的数据传输线连接到 DSP 板子上。
- 将数据传输线的另一端连接到 PC 上正确的接口（USB 或 EPP）。
- 将音频输入设备比如麦克风或者 CD 的输出连接到板子上的音频输入。也可以将 PC 声卡的音频输出直接连接到板子上的音频输入。
- 将一个（或多个）扬声器或其他音频输出设备连接到板子的音频输出端。
- 给板子接通电源。
- 打开 PC。

下面列出运行 RF1 所需要的软件安装步骤。

- 安装 CCS2.2 以上的版本，推荐安装最新的版本。
- 检查并口的配置，确保并口工作在 ECP 或 EPP 模式下，一般为 0x378 端口。检查并口配置的详细资料，参见随板子提供的快速指南。
- 使用 Setup CCS 程序配置板子需要的软件，具体细节参见板子提供的文档。
- 从 TI 网站（www.dspvillage.com）下载 RF 文件，将此文件解压缩。推荐放置于 CCS 安装目录下的 myproject 文件夹。不要将解压的文件放到 c:\Program Files 文件夹。
- RF 解压后的顶层文件夹叫做“referenceframeworks”，在以后章节中这个文件夹的路径用“RF\_DIR”代替。

下一步是运行 genbufs，它是 RF1 中包含的一个缓冲预生成应用程序，用于判定算法的需求，会产生一个能用在目标系统中的源文件，下面是具体步骤。

- 在 CCS 中，选择 Project→Open，然后打开“RF\_DIR\apps\rf1\genbufs\target”路径下的 genbufs.pjt 工程文件，注意选择与 DSP 板子匹配的 target（如 dsk5402）。
- genbufs 程序生成一个名为 mod\_staticBufsnn.c 的源文件，例如 fir\_staticBufs54.c。这个文件会直接用于目标系统中以声明 FIR 算法的数据缓冲需求。
- 选择 Project→Build 以创建 genbufs.out。
- 选择 File→Load Program 以下载 Debug 文件夹中的 genbufs.out 文件到板子上。
- 选择 DSP/BIOS→Message Log 显示 MOD\_xdasBufSrcTrace 日志。
- 在信息日志窗口用右键点击，然后选中 Property Page，按图 2.10 所示修改，生成日志文件（例如 fir\_staticBufs54.c）。注意要选择“Overwrite”以覆盖已经存在的文件，并且不要在“Sequence Numbers in Output”复选框中打勾。

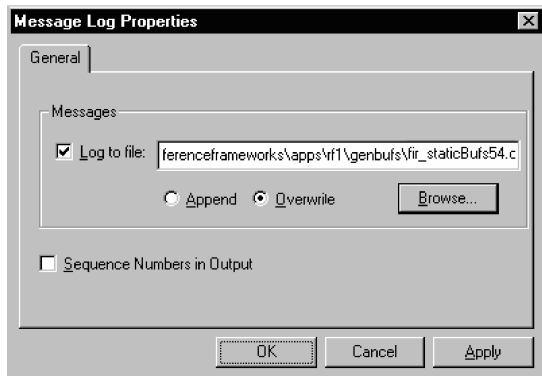


图 2.10 genbufs 应用程序的日志属性

- 选择 Debug→Run。生成的文件会出现在信息日志窗口中。
- 打开 fir\_staticBufs54.c 文件，检查它的内容。声明部分应该与图 2.11 所示类似。
- 下一步 fir\_staticBufs54.c 文件会和 rf1.pjt 工程一起编译。它静态创建算法要求的缓冲区的大小以及对齐方式。也能为可重定位的数据段单独分配缓冲区，以达到最佳的分配和缓冲区的重复使用。由于这个源文件可以有效地、静态地满足算法对缓冲区的需求，从而避免了非常耗内存空间的堆存储器（heap）的动态分配。

```
#pragma DATA_SECTION(firChanBufId00, ".bss:firChanBufId00")
#pragma DATA_ALIGN(firChanBufId00, 4)
#pragma DATA_SECTION(firChanBufId01, ".bss:firChanBufId01")
#pragma DATA_ALIGN(firChanBufId01, 2)
#pragma DATA_SECTION(firChanBufId02Scr, ".bss:firChanBufId02Scr")
#pragma DATA_ALIGN(firChanBufId02Scr, 2)
Char firChanBufId00[6];
Char firChanBufId01[31];
Char firChanBufId02Scr[287];
```

图 2.11 fir\_staticBufs54.c 文件

在生成 mod\_staticBufsnn.c 文件后, 就可以创建和运行 RF1 应用程序了。具体步骤为:

- 在 CCS 中, 选择 Project→Open 打开 RF\_DIR\apps\rf1\target 路径下的工程文件 rf1.pjt, 选择与 DSP 板子匹配的 target (如 RF\_DIR\apps\rf1\dsk5402);
- 例如前面生成的 fir\_staticBufs54.c 文件在这个工程的源文件夹中;
- 选择 Project→Build 创建可执行程序 rf1.out;
- 选择 File→Load Program 将 Debug 文件夹中的 rf1.out 文件下载到目标板上;
- 启动 CD 或是其他的音频输入设备;
- 选择 Debug→Run (或者按 F5), 就会听见从连接在目标板上的喇叭中传出的 FIR 滤波后的音频输出。

### 2.2.2.3 RF1 文件描述

RF1 包括两个 CCS 工程: genbufs.pjt 和 rf1.pjt。如前所述, 先运行 genbufs 工程生成一个用在 rf1.pjt 工程中的源文件 mod\_staticBufsnn.c。rf1.pjt 工程是自己创建的应用程序的基础。

图 2.12 所示为 RF1 中使用的文件夹以及所包含的重要文件。主要的文件夹有以下几个。

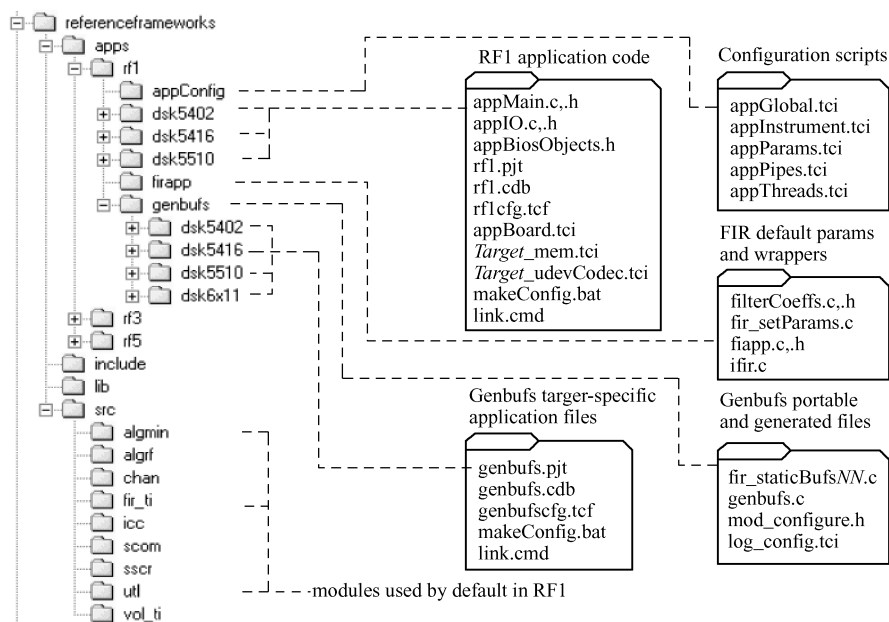


图 2.12 RF1 文件夹



- rf1: 包含 RF1 的两个工程。使用时, 需要复制相应板子的全部文件夹, 然后修改。
- appConfig: 包含 RF1 应用中与目标板独立的配置脚本。
- target: 如 dsk5402, 包含特定目标板的 RF1 代码。
- firapp: 包含与应用程序进行通信的 FIR 算法的封装器。以后自己的算法也应类似地封装, 以便与 RF1 架构相匹配。
- genbufs: 包含一个应用程序, 运行后生成一个在 RF1 应用程序中会用到的资源需求说明源文件。RF 为多种目标板提供了不同的版本。
- include: 包含 RF 公共的头文件, RF1 只使用了其中一部分。另外, 只与一个源代码相关的私有头文件与源文件放到一起, 不放到此文件夹中。
- lib: 包含一些与 RF 链接的库文件。RF1 使用了其中的一些, 但不是全部。
- src: 包含在 include 和 lib 文件夹中的模块的源文件夹。每个文件夹均有 readme.txt 文件, 提供了模块的信息和使用方法。特别是库模块一般不需要改动或是改动很少。在 RF1 中用到的有三个文件夹。
- algmin: 包含了 RF1 中 IALG 实现的源文件, 给出了最小化存储器空间占用的符合标准的算法的例子。
- fir\_ti: 包含了 TI 提供的有限冲激响应滤波器的算法源文件。
- utl: 包含了 UTL 调试诊断模块的源文件。

#### 2.2.2.4 RF1 算法接口

2.2.1 节中已提到, RF1 算法接口为 ALGMIN。由于 RF1 的目的是提供小而有效的模块, 故只用不到 100 行代码实现了 XDAIS 的一个静态实例。图 2.13 为 ALGMIN 函数的调用顺序。其中 ALGMIN\_init 不做任何事情, ALGMIN\_exit 应在算法最后结束时调用, 无返回值。

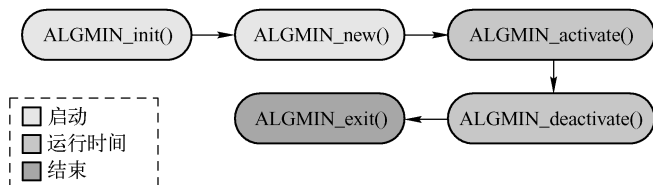


图 2.13 ALGMIN 函数的调用顺序

真正的算法初始化函数是: IALG\_Handle ALGMIN\_new (IALG\_Fxns \* fxns, IALG\_Params \* params, Char \* algChanBufs[], SmUns numAlgChanBufs); 它初始化算法预先配置的缓冲区的大小和对齐方式、运行算法的 algInit 函数、返回算法实例的句柄。

ALGMIN\_activate 和 ALGMIN\_deactivate 函数均无返回值, 只有一个参数: 算法实例的句柄。ALGMIN\_activate 初始化算法实例对象拥有的可擦写和共享静态存储区。ALGMIN\_deactivate 将算法实例对象的状态信息从可擦写缓存保存到非易失静态存储器。

ALGMIN 要求 RF1 用 genbufs 或其他类似应用程序生成算法要求的数据缓冲区的定义文件, 因为算法运行时只使用静态存储, 不使用从堆中动态分配的存储区。应用程序可以简单地把预先生成的缓冲区封装成数组或缓冲区指针。后面将以 FIR 为例详细介绍 RF1 算法。

### 2.2.2.5 RF1 模块结构

RF1 为构造简洁的静态系统提供了方法论。生成一个好的基于 RF1 框架的应用的关键在于选择正确的低级别服务，并且以尽可能少地占用存储空间的方式应用这些服务。

RF1 中用到的 DSP/BIOS 模块有：HWI（硬件中断管理）、IDL（空闲函数管理）、PIP（缓冲管道管理）、ATM（原子函数管理）、CLK（时钟管理）、STS（统计对象管理）、LOG（事件日志管理）。

RF1 中用到的 CSL 模块有：MCBSP（多通道缓冲串口管理）、DMA（直接存储器访问模块）、TIMER（定时器模块）、IRQ（中断控制模块）。

RF1 中用到的 I/O 驱动模块有：IOM（I/O 迷你驱动接口）、PIO（PIP 适配器）。当 RF1 应用程序启动时，主函数初始化各模块以及创建应用程序中要用到的对象。其中数据流对象包括 DSP/BIOS 管道对象（pipRx 和 pipTx）以及 PIO 对象（pioRx 和 pioTx）。图 2.14 简要描述了 RF1 中的数据流路径。

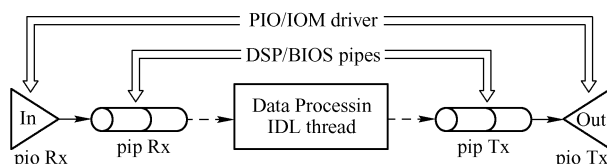


图 2.14 RF1 数据流

为了实现最小的存储器占用，RF1 中 DSP/BIOS 对象均采用静态配置，从而在应用程序中不会出现创建对象（例如：XXX\_create()）的调用，并且 RF1 将数据处理放到 IDL，而不是软中断（SWI）。虽然与任务相比，软中断是轻量级线程，可共享程序堆栈；但是，对于非常紧凑的系统，即使软中断也提供了多余的服务。由于 RF1 中所有的信道和算法都运行于同一优先级，没有必要应用 DSP/BIOS 全部抢占内核的功能。RF1 使用更简单的 IDL 模块，采用 IDL 函数 checkFlags() 持续地检查后台的输入和输出的状态标志，当这两个标志都被设置则触发数据处理函数运行。用 IDL 代替 SWI 可节约 456 字的程序空间。同理，RF1 去掉 RTDX 和 SYS 模块也可以节约存储空间。最优化的 DSP/BIOS 程序和数据存储器仅占用 1.6KB 存储空间。表 2.3 为 RF1 的存储器需求以及对应 DSP 处理器剩余的存储器空间。一个运行于 C54X DSP 的第三方符合算法标准的 MP3 解码器需要的存储器少于 10KB，根据表 2.3，可以采用 RF1 移植到 TMS320VC5402 处理器上。

表 2.3 RF1 的存储器需求

	TMS320VC5402	TMS320VC5510
RF1 存储器需求（代码 + 数据）	4KB	5.6KB
优化后的 RF1 存储器需求	3.5KB	5.0KB
给算法剩余的存储空间	$16 - 4 = 12\text{KB}$	$160 - 5.6 = 154.4\text{KB}$
优化后给算法剩余的存储空间	$16 - 3.5 = 12.5\text{KB}$	$160 - 5 = 155\text{KB}$

### 2.2.2.6 RF1 算法举例——FIR

FIR 滤波器很简单，其数学表达式为：

$$y(k) = \sum_n w(n)x(k-n), k=0,1,\dots \quad (2.1)$$

TI 提供的 FIR 滤波器库是完全符合 XDAIS 标准的算法模块。首先, `algAlloc()` 的函数指针映射到 `FIR_TI_alloc()`, 它需要三个数据缓存区, 其中两个保存永久状态, 一个是可擦写的工作缓存。三个缓存的大小均由输入的参数决定, 例如:

```
/* Request memory for persistent filter history buffer */
memTab[ HISTORY ]. size = ( params -> filterLen - 1 ) * sizeof( Short );
memTab[ HISTORY ]. alignment = 2;
memTab[ HISTORY ]. space = IALG_EXTERNAL;
memTab[ HISTORY ]. attrs = IALG_PERSIST;
/* Request memory for shared working buffer */
memTab[ WORKBUF ]. size = ( params -> filterLen - 1 + params -> frameLen ) * sizeof( Short );
memTab[ WORKBUF ]. alignment = 2;
memTab[ WORKBUF ]. space = IALG_DARAM0;
memTab[ WORKBUF ]. attrs = IALG_SCRATCH;
```

其次, `FIR_TI_initObj()` 对实例对象中的指针赋值, 使它指向框架分配的存储器。接着 `FIR_TI_activate()` 把一小部分滤波器历史值复制到可擦写缓存的起始部分, 这些值主要用于运行主处理函数——`FIR_TI_filter()`, 从而产生滤波后的输出。这就是一个滤波函数, 即将当前输入数据按静态滤波器系数进行低通或高通等滤波。`FIR_TI_deactivate()` 将更新后的滤波器历史数据存回到外部永久缓存区, 以便永久保存。所有全局的标识符都遵循算法标准的规则, 以 `FIR_TI_` 为前缀。

虽然可以通过查阅 TI 的资料来配置 FIR 工程需要的资源, 但运行 `genbufs` 生成一个 RF1 工程中所使用资源 C 文件更简单且不易出错, 过程如图 2.15 所示。图中的 `fir_staticcfg.c` 是生成的结果文件, 这个文件在 RF1 中可以直接保存和加载, 只有当新的信道加入或者参数重新配置时才需要再运行 `genbufs`。

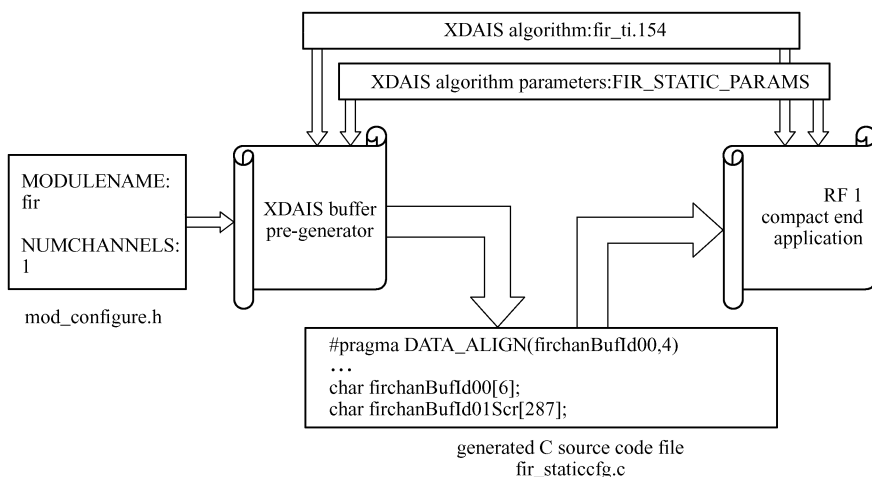


图 2.15 通过 `genbufs` 产生 FIR 的 RF1 配置文件

图 2.16 所示为 FIR 的线程调度图, 输入管道 (`pipRx`) 的标志位表示输入缓冲器已准备

好；输出管道（pipTx）的标志位表示输出缓冲器已准备好。IDL 函数 checkFlags() 反复运行，检查输入输出标志位，来决定什么时候处理数据。如前所述，RF1 没有使用 SWI 和 TSK，只用了优先级最低的 IDL。输入/输出外设产生的硬件中断（HWI）运行在最高的优先级，DSP 空闲时才启动的 IDL 运行在低优先级，IDL 触发进行数据处理的 FIR 算法运行。

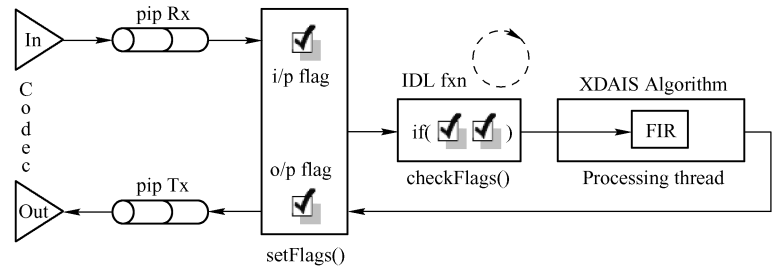


图 2.16 应用 RF1 的 FIR 调度图

FIR 算法包括的模块有：FIR\_init()、FIR\_new()、FIR\_apply()、FIR\_exit()。FIR\_init() 初始化全局的数据分配，通常为一个空函数，但仍有必要存在，它使 RF1 执行一个预先定义好的启动序列。FIR\_new() 为算法模块创建一个完整的实例，通常存储器已经分配好了，在这个函数中只需要进行将指针指向已分配的存储器。实际上，FIR\_new() 只是简单地调用 ALGMIN\_new()，并返回句柄。使用了 firChanBufs 队列，以便算法能在 ALGMIN\_new() 函数中决定缓冲器的数量。这样如果需要修改，就不用在 FIR\_new() 函数中修改缓冲器表，只需在 appMain.c 中对 firChanBufs 队列进行声明即可。FIR\_apply() 执行滤波器算法。TI 提供的 FIR 算法使用了存储器保存数据，因此要求 RF1 调用可擦写存储器操作函数 algActivate() 和 algDeactivate()。FIR\_exit() 进行模块结束操作，此版本暂时没有这一函数，留待以后扩充。

如前所述，RF1 应用的一个主要目标是实现最小的存储器占用，在 FIR 应用中，首先关闭 RF1 用不到的 DSP/BIOS 模块和驱动模块，如 TSK、SWI、PRD、RTDX、SYS、DSP 负载计算等；其次用配置工具静态配置存储器等参数；再采用优化技术减少代码和数据的存储器占用，如创建虚拟段映射没有用到的 XDAIS IALG 函数，采用 UNION 指令将多个段分配到同一地址，同一算法的多个信道可以重复使用同一块可擦写缓存，覆盖技术可以把只运行一次的安装代码使用的空间重新利用等。表 2.4 所示为 TMS320VC5510DSK 上运行基于 RF1 的 FIR 算法的详细存储器需求，其中用于调试的部分可以进一步去掉或减少。更深入的降低存储器的优化技术请见参考文献 [6]。

表 2.4 FIR 的存储器需求

总 和	12 667 字 (16 比特)	总 和	12 667 字 (16 比特)
缓存	1 024	UTL 对象 (调试用)	151
算法使用的堆	324	. cinit	1 351
FIR 算法	261	AlgMin 覆盖段	36
FIR 系数表	32	CSL 覆盖段	1 250
系统栈	1 914	驱动覆盖段	175
LOG 对象 (调试用)	173	PIO 覆盖段	168
STS 对象 (调试用)	35		

### 2.2.2.7 将 RF1 应用于其他算法

下面以 G726 编解码器为例，描述如何应用 RF1 实现自己的算法。采用装载了 FIR 算法的 RF1 为模板，把 FIR 算法用 G726 算法替代即可，如图 2.17 所示。具体的配置步骤为：

- 创建一个应用文件夹；
- 复制算法文件到公共的库和头文件夹中；
- 建立算法应用文件夹；
- 修改和运行 genbufs. pj1 工程；
- 修改 rf1. pj1 工程。

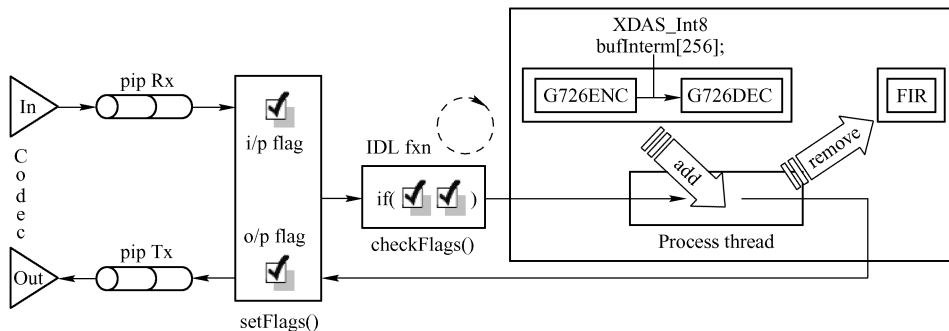


图 2.17 RF1 中将 FIR 替换成 G726

#### 1) 创建应用文件夹

这一步，推荐用户最好复制整个应用文件夹，以便可以参考和保留不需要修改的内容。例如，把图 2.12 中的“apps\rf1”目录树整个复制，并重命名为“rf1\_g726”。

下面的文件均在“RF\_DIR\apps\rf1\_g726”这个目录结构中，而且文件名中，mod 代表算法名字的小写，MOD 代表算法名字的大写。

#### 2) 复制算法文件

与其他 XDAIS 算法相同，G726 编解码器也包含四个部分：库、头文件、函数封装和默认参数。为了简化这些文件的路径，将库放置在公共的库文件夹中，头文件在公共的头文件夹中，其他文件放在本地应用文件夹中。

TI 提供的 G726 编解码器算法的文件放在 TI\_DIR\examples\target\xdais\demo\lib 和 TI\_DIR\cXXXX\xdais\src\vocoders 文件夹中。故需要拷贝的文件有以下几类。

- 库文件

g726enc\_ti. l54 和 g726dec\_ti. l54；

从 TI\_DIR\examples\target\xdais\demo\lib 复制到 RF\_DIR\lib。

- 头文件

g726enc\_ti. h 和 g726dec\_ti. h；

从 TI\_DIR\examples\target\xdais\demo\include 复制到 RF\_DIR\include

ig726. h、ig726enc. h 和 ig726dec. h；

从 TI\_DIR\c5400\xdais\src\vocoders 复制到 RF\_DIR\include。

#### 3) 建立算法应用文件夹

在 rf1\_g726 文件夹中，建立一个 g726app 文件夹，来存放算法函数封装和默认参数

文件。

- 复制下列源文件，它们指定了算法的默认参数。

ig726enc.c 和 ig726dec.c；

从 TI\_DIR\c5400\xdais\src\vocoders 复制到 RF\_DIR\apps\rf1\_g726\g726app。

- 复制下列源文件和头文件，并重命名。

fir\_setParams.c 重命名为 g726enc\_setParams.c。

firapp.c 重命名为 g726encapp.c。

firapp.h 重命名为 g726encapp.h。

从 RF\_DIR\apps\rf1\firapp 复制到 RF\_DIR\apps\rf1\_g726\g726app。

- 再额外复制一组文件，和上一步骤类似，只是重命名时用 g726dec 代替 g726enc。最后得到的文件夹和文件如图 2.18 所示。

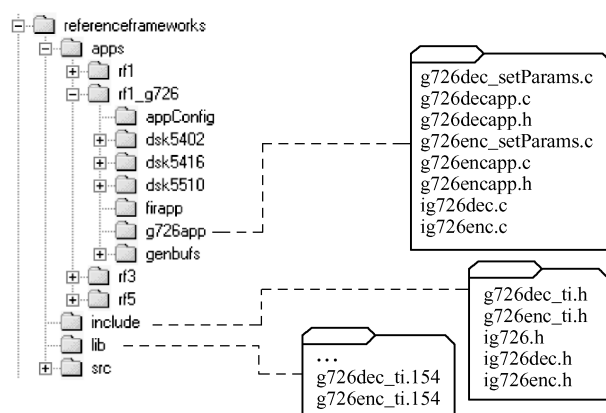


图 2.18 RF1 实现 G726 文件夹

- 注意在 CCS 中要完成 g726app 文件夹中所有文件内容的搜索和替换，即：FIR 替换成 G726ENC 或 G726DEC，fir 替换成 g726enc 或 g726dec。
- 对 g726enc\_setParams.c 文件做如下改变：
  - 去掉#include " filterCoeffs.h"；
  - 打开 imod.c 文件，然后复制 MOD\_Params 静态初始化里的参数，如从 ig726enc.c 文件中复制粗体文本到剪贴板；

```
const IG726ENC_Params IG726ENC_PARAMS = {
    sizeof(IG726ENC_PARAMS),          /* Size of this structure */
    1,                                /* Sample by sample processing */
    IG726_ALAW,                        /* Input buffer format is A-law */
    IG726_16KBPS,                      /* Working rate is 16kbps */
};
```

- 粘贴复制得文本到 mod\_setParams.c 里，如下面粗体显示的；

```
IG726ENC_Params G726ENC_STATIC_PARAMS[ ] = {
    {
```



```

        sizeof(IG726ENC_PARAMS),      /* Size of this structure */
        1,                            /* Sample by sample processing */
        IG726_ALAW,                   /* Input buffer format is A-law */
        IG726_16KBPS,                 /* Working rate is 16kbps */
    },
    /* + ENTER NEW CHANNEL PARAMETER SETS BELOW + */
}; /* end of G726ENC_STATIC_PARAMS[] */

```

- 根据需要改变 mod\_setParams. c 文件里的算法参数。

如 TI 的 G. 726 例子中，粗体表示修改的部分：第二个参数从 1 改为 256；第三个参数从 IG726\_ALAW 改变为 IG726\_LINEAR。

```

IG726ENC_Params G726ENC_STATIC_PARAMS[] = {
    {
        sizeof(IG726ENC_PARAMS),      /* Size of this structure */
        256,                          /* Process by frames instead of by sample */
        IG726_LINEAR,                 /* Input buffer format is LINEAR */
        IG726_16KBPS,                 /* Working rate is 16kbps */
    },
    /* + ENTER NEW CHANNEL PARAMETER SETS BELOW + */
};

```

- 与上一步类似，对 g726dec\_setParams. c 文件进行修改。

#### 4) 修改和运行 genbufs. pj1 工程

使用 genbufs，生成定义每个算法需要的数据缓冲区的 C 源文件。

- 在 CCS 中，打开工程文件 RF\_DIR\apps\rf1\_g726\genbufs\target\genbufs. pj1。每个算法都需要执行一遍下面的步骤。例如，G. 726 编码器和解码器必须执行两次以下步骤。
- RF\_DIR\apps\rf1\_g726\genbufs 文件夹中做下列的操作。
  - genbufs\mod\_configure. h 中将 MODULENAME 常数定义成算法的名字，对于 G. 726 编码器的定义用粗体显示为：

```
#define MODULENAME g726enc
```

- genbufs\target\ link. cmd 文件中进行区分大小写的搜索和替换，对于 G. 726 编码器，将 FIR 替换成 G726ENC，将 fir 替换成 g726enc。
- 在 CCS 中，从第三步建立的算法应用文件夹（RF\_DIR\apps\rf1\_g726\g726app）添加下列的源文件到 genbufs 工程中，见表 2. 5。

表 2. 5 添加文件到 genbufs

需添加的文件	添加到 G. 726 编码器	添加到 G. 726 解码器
mod_setParams. c	g726enc_setParams. c	g726dec_setParams. c
imod. c	ig726enc. c	ig726dec. c



- 移除 genbufs 工程的下列源文件。

```
filterCoeffs. c
fir_setParams. c
ifir. c
```

- 运行 genbufs 应用程序，输出文件写到 genbufs\mod\_staticBufsnn. c（例如，g726enc\_staticBufs54. c），需要为每个算法产生的文件使用不同的名字。

#### 5) 修改 rf1. pj1 工程

按照以下列步骤，修改 RF1 应用程序。

- 在 CCS 中，打开工程文件 RF\_DIR\apps\rf1\_g726\target\rf1. pj1。
- 在 CCS 中，选择 Project→Build 选项。在 Compiler 选项卡中，选择 Preprocessor，将头文件的搜寻路径里的 firapp 改为 g726app。
- 移除 rf1. pj1 工程中的下列源文件。

```
filterCoeffs. c
fir_setParams. c
firapp. c
ifir. c
fir_staticBufsnn. c
```

- 从 modapp\文件夹中把下面的源文件加入 rf1. pj1 工程中，见表 2.6。

表 2.6 添加文件到 rf1. prj

需添加的文件	添加到 G.726 编码器	添加到 G.726 解码器
mod_setParams. c	g726enc_setParams. c	g726dec_setParams. c
imod. c	ig726enc. c	ig726dec. c
modapp. c	g726encapp. c	g726decapp. c

- 从 RF\_DIR\apps\rf1\_g726\genbufs 文件夹中把下列的文件添加到 rf1. pj1 工程中，这些文件是 genbufs 应用产生的输出文件。

```
g726enc_staticBufsnn. c
g726dec_staticBufsnn. c
```

- 打开每个算法的 RF\_DIR\apps\rf1\_g726\genbufs\mod\_staticBufsnn. c 文件和 RF\_DIR\include\imod. h 文件，以便编辑其他的文件时候，能看到它们的内容。
- 对每个 modapp\modapp. h 文件（例如：g726encapp. h），做下列改变。
  - 修改信道缓冲器的外部声明以匹配 mod\_staticBufsnn. c 文件，例如，从 FIR 算法转换为 G726ENC 时，需移除可擦写缓冲器的外部声明。
  - 在 MOD\_apply 函数定义中，改变输入和输出指针，以匹配 imod. h 文件中定义的 MOD\_Fxns 结构。例如，ig726enc. h 中编码函数有如下声明：

```
XDAS_Int16 ( * encode)( IG726ENC_Handle handle, XDAS_Int16 * in, XDAS_Int8 * out );
```

于是, 在 g726encapp.h 中, 进行下列各项修改 (见黑体), 以便从 FIR 转换为 G726ENC。

```
/*
 * ===== G726ENC_apply =====
 * Apply G726ENC algorithm to input array and place results in output array.
 */
extern Void G726ENC_apply( G726ENC_Handle handle, Short in[ ], Short out[ ] );
extern Void G726ENC_apply( IG726ENC_Handle handle, XDAS_Int16 * in, XDAS_Int8 * out );
```

- 对每个 modapp\modapp.c 文件, 做下列改变。
  - 在 MOD\_apply 函数定义中, 改变输入和输出指针, 以匹配 MOD\_Fxns 结构的原型。

```
/* ===== G726ENC_apply =====
 * Apply G726ENC algorithm to input array and place results in output array. */
Void G726ENC_apply( G726ENC_Handle handle, Short in[ ], Short out[ ] )
Void G726ENC_apply( IG726ENC_Handle handle, XDAS_Int16 * in, XDAS_Int8 * out )
```

- 在 MOD\_apply 函数中, 改变算法指针, 使算法成为 RF\_DIR\include\imod.h 文件中定义的 IMOD\_Fxns 结构中的一个操作。例如, 从 FIR 算法转换为 G726 ENC 时按下面的方式修改, 修改后的代码见黑体。

```
handle -> fxns -> filter( handle, in, out ); /* filter data */
handle -> fxns -> encode( handle, in, out ); /* encode data */
```

- 对每个算法都要按上述步骤进行修改。对于本例, 还需为 G. 726 解码器重复上述步骤。
- 下面修改定位文件 target\link.cmd (例如, dsk5402\link.cmd)。
- 连接库文件时去掉 FIR 算法库, 加入 G. 726 算法库。应该为每个算法连接单独的库, 例如, 分别与 G. 726 编码器和解码器库连接, 需要进行如下修改。

```
/* XDAIS algorithm(s) */
-l fir_ti.l54
-l g726enc_ti.l54
-l g726dec_ti.l54
```

- 要使用 TI 提供的 G. 726 编码器和解码器, 请参考 RF\_DIR\include\g726enc\_ti.h 和 g726dec\_ti.h 文件, 需进行如下修改。

```
/* Point to the algorithm vendor's IMOD v-table (name located in mod_ven.h) */
_FIR_IFIR = _FIR_TI_IFIR;
_G726ENC_IG726ENC = _G726ENC_TI_IG726ENC;
_G726DEC_IG726DEC = _G726DEC_TI_IG726DEC;
```

- 修改段的第一个列表, 以匹配 mod\_staticBufsnn.c 文件中非可擦写缓冲器定义。例

如, 使用 TI 的 G. 726 编码器和解码器需进行如下修改。

```
SECTIONS
{
    .bss:firChanBufId00: fill = 0x0 {} > IDATA PAGE 1
    .bss:g726encChanBufId00: fill = 0x0 {} > IDATA PAGE 1
    .bss:g726decChanBufId00: fill = 0x0 {} > IDATA PAGE 1
} /* end of persistent buffer fill section */
```

- 写缓冲器 (algChanBufIdnnScr) 相匹配。例如, TI 的 G. 726 编码器和解码器没有使用可擦写缓冲器, 因此, 只是简单的删除声明并将 UNION 注释即可。
- 接下来修改 target\appMain.c。
- 首先在这个文件上运行一次区分大小写的搜索和替换操作, 将所有 FIR 改为 G726ENC; 所有 fir 改为 g726enc。
- 添加算法的声明头文件, 例如: 增加 G. 726 编解码器算法。

```
#include "g726encapp.h" /* algorithm' s application interface(s) */
#include "g726decapp.h" /* algorithm' s application interface(s) */
```

- 增加声明使算法可以实例化, 例如, G. 726 编解码器算法。

```
/* Declaration of XDAIS v-tables. These are the algorithm(s) entry pts */
extern IG726ENC_Fxns G726ENC_IG726ENC; /* G726ENC algorithm */
extern IG726DEC_Fxns G726DEC_IG726DEC; /* G726DEC algorithm */
/* Handles to the XDAIS algorithm(s). Provides reentrant access to the fxns */
G726ENC_Handle g726encHandle0;
G726DEC_Handle g726decHandle0;
```

- 在主函数的开始定义的 modChanBufs 数组中, 改变缓冲器的数量以便与算法中用到的数量相匹配。例如, G. 726 编码器删除了 FIR 用到的第三个缓冲器。

```
/* Prepare buffer pointer array for Static XDAIS Algorithm instantiation */
Char *g726encChanBufs[] = { g726encChanBufId00, g726encChanBufId01, g726encChanBufId02Scr };
```

- 为任何增加的算法添加新的 modChanBufs 定义。例如, 为 G. 726 解码器加入下列语句。

```
Char *g726decChanBufs[] = { g726decChanBufId00, g726decChanBufId01 };
```

- 在主函数中, 为任何增加的算法增加初始化函数调用。例如, 为 G. 726 解码器算法增加粗体的行。

```
/* Initialize the XDAIS algorithm(s) */
G726ENC_init();
G726DEC_init();
```

- 为任何增加的算法加入 MOD\_new 的调用。例如，为 G. 726 解码器增加下列调用。

```
/* Statically create an instance of the XDAIS algo(s) ie no heaps etc */
g726encHandle0 = G726ENC_new( &G726ENC_IG726ENC, G726ENC_chanParamPtrs[0],
g726encChanBufs, ( sizeof( g726encChanBufs ) / sizeof( g726encChanBufs[0] ) ) );
g726decHandle0 = G726DEC_new( &G726DEC_IG726DEC, G726DEC_chanParamPtrs[0],
g726decChanBufs, ( sizeof( g726decChanBufs ) / sizeof( g726decChanBufs[0] ) ) );
```

- 在 audioProcess 函数中，将源和目的数据类型换成与 MOD\_apply 函数参数声明的类型相匹配。并为任何增加的算法添加 MOD\_apply 的调用。例如，使用 G. 726 编码器和解码器算法时需作如下修改。

```
/* Execute the XDAIS Algorithm(s) */
G726ENC_apply( g726encHandle0, ( ShortXDAS_Int16 * )src, ( ShortXDAS_Int8 * )dst );
G726DEC_apply( g726decHandle0, ( XDAS_Int8 * )src, ( XDAS_Int16 * )dst );
```

- 在主函数之前，加上如下的全局声明以创建一个中间缓冲区，用于存储编码器的输出。

```
XDAS_Int8 buffInterm[256]; /* intermediate buffer to store encoder output */
```

- 在 audioProcess 函数中，增加下面的语句，使用了这个中间缓冲区。

```
/* Execute the XDAIS Algorithm(s) */
G726ENC_apply( g726encHandle0, ( XDAS_Int16 * )src, ( XDAS_Int8 * )dstbuffInterm );
G726DEC_apply( g726decHandle0, ( XDAS_Int8 * )srcbuffInterm, ( XDAS_Int16 * )dst );
```

- 保存所有修改过的文件和工程。
- 创建并运行 RF1 应用程序。这时听到的声音与 FIR 滤波器处理后的声音相比，有一点失真。失真通常是由声码器的压缩率造成的。如果一点声音都没听到，需在 g726enc\_setParams.c 和 g726dec\_setParams.c 文件中检查 G726ENC\_STATIC\_PARAMS 和 G726DEC\_STATIC\_PARAMS 等。

### 2.2.2.8 RF1 增加第二个信道

RF1 虽然为只使用少量信道的应用而设计。许多简单的应用可能会有使用第二个信道的需要。例如，立体声 MP3 播放机就有左、右信道。

因为基本的 RF1 应用程序是一个单一程序。下面这个例子将一个信号复制到两个通道中，从而不用改变应用程序的逻辑就可以实现真正的立体声。现在假设以基本的 FIR 滤波器为例，将单声道修改为左右声道均可使用，每个声道可以采用不同的静态参数。

下面的文件路径，RF\_DIR 表示“参考工作”文件夹。首先需要创建与它相关的其他文件路径（比如 RF\_DIR\apps\rf1\_2ch）。具体步骤如下：

- 1) 在 apps\文件夹中，复制整个 rf1\目录树，并将复制的文件夹命名为“rf1\_2ch”。
- 2) 在 CCS 中，打开工程文件 RF\_DIR\apps\rf1\_2ch\genbufs\target\genbufs.pjt。

3) 在 `genbufs\mod_configure.h` 文件中, 把 `NUMCHANNELS` 常数修改为实际要使用的信道数量。在这个例子中, 将 `NUMCHANNELS` 设置为 2。

```
#define MODULENAME fir
#define NUMCHANNELS 2
```

4) 如果所有信道的算法参数都一样, 就直接跳至步骤 5)。若第二个信道使用不同的算法参数, 则需要编辑 `firapp\fir_setParams.c` 文件, 加入一个信道参数块到 `FIR_STATIC_PARAMS` 中, 以改变参数。

第二信道的典型修改就是使用不同的滤波器系数。例如, 首先在 `firapp\filterCoeffs.c` 和 `firapp\filterCoeffs.h` 中定义一个名为 `filterCoeffs_1` 的常量数组, 存放新的滤波器系数, 再按照下面粗体语句增加第二个信道的参数设置。

```
IFIR_Params FIR_STATIC_PARAMS[ ] =
{
    {
        sizeof(IFIR_Params), /* sizeof base params structure */
        (short *)filterCoeffs, /* coefficient array */
        COEFFSIZE, /* filter length */
        FILTERFRAMESIZE, /* frame length */
    },
    /* + ENTER NEW CHANNEL PARAMETER SETS BELOW + */
    {
        sizeof(IFIR_Params), /* sizeof base params structure */
        (short *)filterCoeffs_1, /* coefficient array */
        COEFFSIZE, /* filter length */
        FILTERFRAMESIZE, /* frame length */
    },
}; /* end of FIR_STATIC_PARAMS[ ] */
```

5) 在 `firapp\fir_setParams.c` 中, 为每个信道添加参数集的起始地址。

- 如果两个信道的参数是一样的, 依照下面例子添加粗体显示的一行。

```
IFIR_Params *FIR_chanParamPtrs[ ] =
{
    &FIR_STATIC_PARAMS[0],
    /* + UPDATE PARAM POINTERS TO REFLECT NEW CHAN PARAM SET ADDITIONS + */
    &FIR_STATIC_PARAMS[0],
};
```

- 如果两个信道的参数不一样, 使用 `&FIR_STATIC_PARAMS[1]` 设置第二组参数地址。

6) 生成并运行 `genbufs` 程序。将输出文件写入 `fir_2ch_staticBufsnn.c` (这是推荐的文件名, 但不是强制要求)。

7) 在 CCS 中打开 `RF_DIR\apps\rf1_2ch\target\rf1.pjt` 工程, 进行以下修改。

- 从工程的源文件夹中移除 fir\_staticBufsnn. c file 文件，添加上一步用 genbufs 生成的文件（例如 fir\_2ch\_staticBfs54. c）到工程中。
- 修改 appMain. c：增加粗体的文本创建 FIR 对象，在第二个信道上运行算法，且算法使用 fir\_2ch\_staticBufsnn. c 中声明的缓冲区。注意每个信道缓冲的命名规律。

另外，这个例子只是简单地对同一个输入缓冲运行两次算法。而真正的立体声应用不是这样，一般应对每个信道使用独立的输入和输出缓冲器。

```

/* Declaration of XDAIS v-tables. These are the algorithm(s) entry pts */
extern IFIR_Fxns FIR_IFIR; /* FIR algorithm */

/* Handles to the XDAIS algorithm(s). Provides reentrant access to the fxns */
FIR_Handle firHandle0;
FIR_Handle firHandle1;
...
/* Prepare buffer pointer array for Static XDAIS Algorithm instantiation */
Char * firChanBufs 0 [ ] = { firChanBufId00, firChanBufId01, firChanBufId02Scr };
Char * firChanBufs1 [ ] = { firChanBufId10, firChanBufId11, firChanBufId12Scr };
...
/* Statically create an instance of the XDAIS algo(s) ie no heaps etc */
firHandle0 = FIR_new( &FIR_IFIR, FIR_chanParamPtrs[ 0 ],
firChanBufs 0, ( sizeof( firChanBufs 0 ) / sizeof( firChanBufs 0 [ 0 ] ) ) );
firHandle1 = FIR_new( &FIR_IFIR, FIR_chanParamPtrs[ 0 ],
firChanBufs1, ( sizeof( firChanBufs1 ) / sizeof( firChanBufs1 [ 0 ] ) ) );
...
/* algorithm execution */
FIR_apply( firHandle0, ( Short * ) src, ( Short * ) dst );
FIR_apply( firHandle1, ( Short * ) src, ( Short * ) dst );

```

如果使用有立体声编解码器的板子，比如 C5416 DSK，则可能需要在 audioProcess() 函数中修改应用逻辑。立体声本身不是基本 RF1 应用的一部分，因为 RF1 的目的是支持非常简洁的消费产品，因此 RF1 设计不是通用和灵活的。

有很多方法可以实现立体声信道，可以通过修改应用程序来实现不同的应用需要，能实现什么应用还与采用的编解码器、算法和驱动等相关。不同的编解码器有不同长度的采样位。立体声编解码器可以实现左边和右边信道为不同的比特数或采样值。算法可能只支持单一信道，或者可以把输入分成左、右信道，或者支持左、右信道交替输入。还要选择驱动，以实现分离输入立体声编解码器的左、右信道。

- 在 firapp\firapp. h 中添加粗体显示的行，声明已在 fir\_2ch\_staticBufsnn. c 中创建的缓冲。

```

/* Match the buffer references to those in the pre-generated buffers C file */
extern Char firChanBufId00 [ ];
extern Char firChanBufId01 [ ];
extern Char firChanBufId02Scr [ ];
extern Char firChanBufId10 [ ];

```

```
extern Char firChanBufId11[];
extern Char firChanBufId12Scr[];
```

- 修改 link.cmd，添加粗体显示的行。

```
SECTIONS
{
    .bss;firChanBufId00; fill = 0x0 {} > IDATA PAGE 1
    .bss;firChanBufId10; fill = 0x0 {} > IDATA PAGE 1
}
/* end of persistent buffer fill section */
...
SECTIONS
{
    UNION: run = IDATAOVL PAGE 1
    {
        GROUP: {
            .bss;firChanBufId02Scr; {}
        }
        GROUP: {
            .bss;firChanBufId12Scr; {}
        }
    }
}
/* end of scratch buffer overlay section */
```

#### 8) 创建并运行 RF1 应用。

至此，我们在 RF1 加上了第二个信道。整个步骤可以作为基于 RF1 应用中添加额外信道的模板。

### 2.2.3 RF3——灵活型编程框架

RF3 的目的在于：帮助设计者实现包括多个算法和通道的简洁产品。这类系统的存储器占用大小是一个需要重点考虑的问题，但同时灵活性也需要关注。

默认情况下，RF3 以给定的采样率将输入的音频信号转换为数字。首先将输入信号分成两个通道，然后独立地处理两个通道，左边通道加一个低通滤波器，右边通道加一个高通滤波器；接着对每个通道分别进行音量控制；最后输出到音频编解码器。在运行时，可以选择每个通道的音量。这一默认的应用提供了使用 RF3 的简单示例，用户很容易根据自己的应用逻辑进行修改。

#### 2.2.3.1 RF3 概述

RF3 的设计目的是方便使用。让一个有丰富应用经验但对 DSP 了解较少的设计者花较少的时间就能构建出系统。符合 eXpressDSP 标准的算法能很容易地集成到 RF3 中。RF3 包



括基础软件，如 DSP/BIOS、片上支持库（CSL），以及 XDAIS。也调用一些参考框架模块提供的服务，例如，UTL 模块用于调试和诊断。开发者可以以现有框架为基础，或以相对简单的方式将自己的应用与更高级别的框架接口。

RF3 比只关注最小存储器占用的 RF1 提供更多的灵活性。但是，RF3 仍然不使用需消耗较多存储器资源的技术，如动态对象创建或线程阻塞等。所以，需要动态创建线程或线程间有复杂关联的应用不适合采用 RF3。

适合使用 RF3 的典型应用为：要求相对较小的存储器占用，但并不严格限制最小的存储器占用；而且，要求比 RF1 提供更强的灵活性。然而，对象和数据通道在运行时均是静态的。也就是说，所有处理线程在程序初始化时被创建，并不能删除。表 2.7 详细给出了 RF3 的特点，可以作为判断 RF3 是否适合目标应用的指南。

表 2.7 RF3 应用的特点

设计参数	RF3	注 释
静态配置	√	
动态对象创建	×	
静态存储管理	√	
动态存储分配	√	
1 ~ 10 通道/信道	√	
11 ~ 100 通道/信道	×	RF3 没有限制通道数目，但 RF5 提供的机制更适合高通道数的应用
1 ~ 10eXpressDSP 算法	√	
11 ~ 100eXpressDSP 算法	×	RF3 没有限制算法数目，但 RF5 提供的机制更适合包括几十个算法的应用
DSP/BIOS 实时分析	√	
DSP/BIOS 内核调度	√	
片上支持库	√	
XDAIS 算法标准	√	
支持多种运行速率和优先级	√	每个通道以单一速率运行，不同数据流可以以不同速率处理
线程阻塞	×	
实现控制函数	√	一个单独的线程或一组线程，可以根据用户的操作改变信号处理的参数

RF3 有以下一些关键特征。

- 消耗与灵活性的最佳折中。RF3 比 RF5 占用更少的存储器，又比 RF1 更灵活和更容易扩展。
- 最易使用。RF3 的一个关键的目标就是通过减小复杂度来达到易于使用。RF3 不包括 RF1 中实现最小化存储器占用的技巧，也不像 RF5 那样支持大数目的算法和通道。
- 基于软件中断（SWI）。RF5 是基于任务（TSK）的，TSK 模块有更好的时序灵活性，但是随之而来的是需要更多的资源消耗和存储器。
- 基于 PIP 的 I/O。相反，RF5 是基于 SIO 的。PIP 模块虽然比 SIO 简单和缺少灵活性，但其优点就是消耗减少。
- 允许简单调试。RF3 定义良好的结构允许快速调试。而且，UTL 模块也可以快速地改变调试级别。
- 可以方便地替换 I/O 驱动。IOM 模块允许将多种迷你驱动连接到 RF3 使用的 PIO 适

配器上。

➤ 支持许多种类的板子。RF3 通常比其他框架支持更多种类的板子。

RF3 的应用系统很广泛,图 2.19 所示为一些 RF3 应用的例子。包括指纹识别、数字无线电、多通道电话、数字播放器、数码相机、数码摄像机、音频增强等。

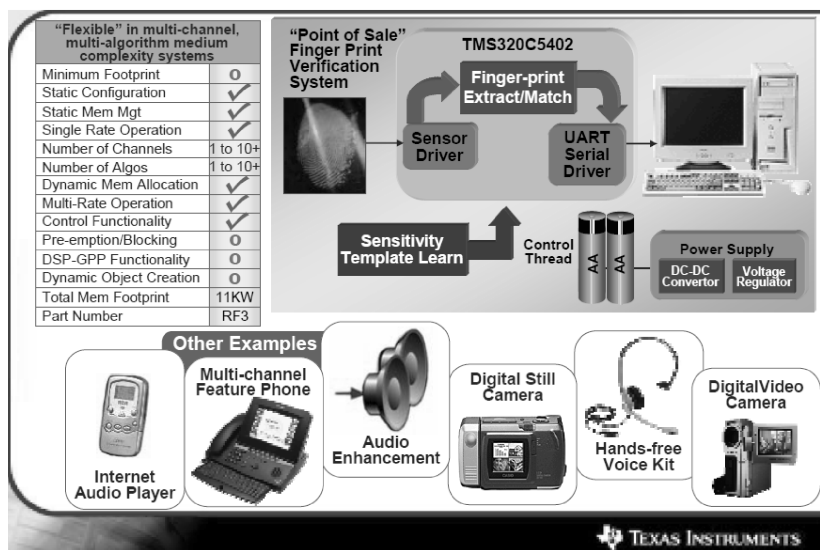


图 2.19 RF3 的应用系统

### 2.2.3.2 安装和运行 RF3

RF3 应用可以与许多板子接口。支持的目标板包括 C5402 DSK、C5416 DSK、C5510 DSK、C6x11 DSK、C6416 等。如果想在其他目标板上运行 RF3, 需要移植依赖硬件的接口部分。

下面的步骤说明怎样将主机与硬件连接。

- 关闭你的 PC。
- 将正确的数据传输线连接到 DSP 板子上。
- 将数据传输线的另一端连接到 PC 上正确的接口 (USB 或 EPP)。
- 将音频输入设备, 比如麦克风或者 CD 的输出连接到板子上的音频输入。也可以将 PC 声卡的音频输出直接连接到板子上的音频输入。
- 将一个 (或多个) 扬声器或其他音频输出设备连接到板子的音频输出端。
- 给板子接通电源。
- 打开 PC。

下面列出运行 RF3 所需要的软件安装步骤。

- 安装 CCS2.2 以上的版本, 推荐安装最新的版本。
- 检查并口的配置, 确保并口工作在 ECP 或 EPP 模式下, 一般为 0x378 端口。检查并口配置的详细资料, 参见随板子提供的快速指南。
- 使用 Setup CCS 程序配置板子需要的软件, 具体细节参见板子提供的文档。
- 从 TI 网站 (www.dspvillage.com) 下载 RF 文件, 将此文件解压缩。推荐放置于 CCS 安装目录下的 myproject 文件夹。不要将解压的文件放到 c:\Program Files 文件夹。

- RF 解压后的顶层文件夹叫作 “referenceframeworks”，在以后章节中这个文件夹的路径用 “RF\_DIR” 代替。

创建和运行 RF3 应用程序的步骤如下。

- 在 CCS 中，选择 Project→Open 打开 RF\_DIR\apps\rf3\target 路径下的工程文件 app.pjt，选择与 DSP 板子匹配的 target（比如，RF\_DIR\apps\rf3\dsk5402）。
- 选择 Project→Build 创建 RF3 可执行程序 app.out。

**注意：**如果使用新版本的 CCS，请在 MS - DOS 命令窗口运行 RF\_DIR\build.bat 这个批处理文件，重建所有的 RF 工程，以确保所有的模块与最新的 TI 代码生成工具同步。

- 选择 File→Load Program 将 Debug 文件夹中的 app.out 文件装载到目标板上。
- 启动 CD 或是其他的音频输入设备。
- 选择 Debug→Run（或者按 F5），就会听见从连接在目标板上的喇叭中传出的 FIR 滤波后的音频输出。
- 选择 File→Load GEL，从工程文件夹中选中 app.gel 文件。app.gel 文件是显示 GUI 控制器的 GEL 脚本。当移动这些控制器时，脚本对目标存储器写入特定的值。这些值控制激活两个通道中的哪一个，以及每个通道的音量级别。
- 选择 GEL→Application Control→Set Active Channel。会出现一个有两个位置的滑动条，用来选择输出通道。下方位置表示选择通道 0，使用一个低通滤波器。上方位置表示选择通道 1，使用一个高通滤波器，它可以使音乐听起来更好。
- 选择 GEL→Application Control→Set\_channel\_0\_gain。出现一个通道 0 音量控制滑动条，音量值的范围为 0 ~ 200，默认值是 100。通道 0 必须被激活，此音量控制改变才能起作用。
- 同样，要控制通道 1 的音量，选择 GEL→Application Control→Set\_channel\_1\_gain，使用滑动条来控制通道 1 的音量。

### 2.2.3.3 RF3 文件描述

RF 的目录树包括应用源文件和库模块。图 2.20 显示了 RF3 的文件夹结构，其中专门标出了一些重要的文件。主要的文件夹包括以下文件。

- apps\rf3 包含了组成 RF3 的 CCS 工程。如果要修改 RF3，可以在同级文件夹下复制整个 rf3\目录树，然后在这个副本上修改。包括的子文件夹有：
  - appConfig，包含与硬件无关的 RF3 配置脚本文件；
  - appModules，包含与硬件无关的 RF3 应用程序，包括线程的实现和初始化代码；
  - target，包含硬件相关的 RF3 应用程序文件，其中有配置文件、工程文件和连接器文件，这些文件放置在硬件平台名字的文件夹中，如 DSK5510 等；
  - algFIR，包含用于与 FIR 算法通信的应用端封装代码；
  - algVOL，包含用于与 VOL 通信的应用端封装代码。
- include 包含在 RF 使用的一些公共头文件。RF3 使用其中一些，但不是全部。算法和框架代码都要引用公共头文件。相反，私有头文件和源代码一起储存，这些源代码包含它们并且不会被其他模块使用。每个库模块在这个文件夹中都有一个头文件。
- lib 包含一些和 RF 应用连接的库文件。RF3 使用其中一些，但不是全部。每个库模块在这个文件夹中都有和 DSP 芯片对应的一个库。

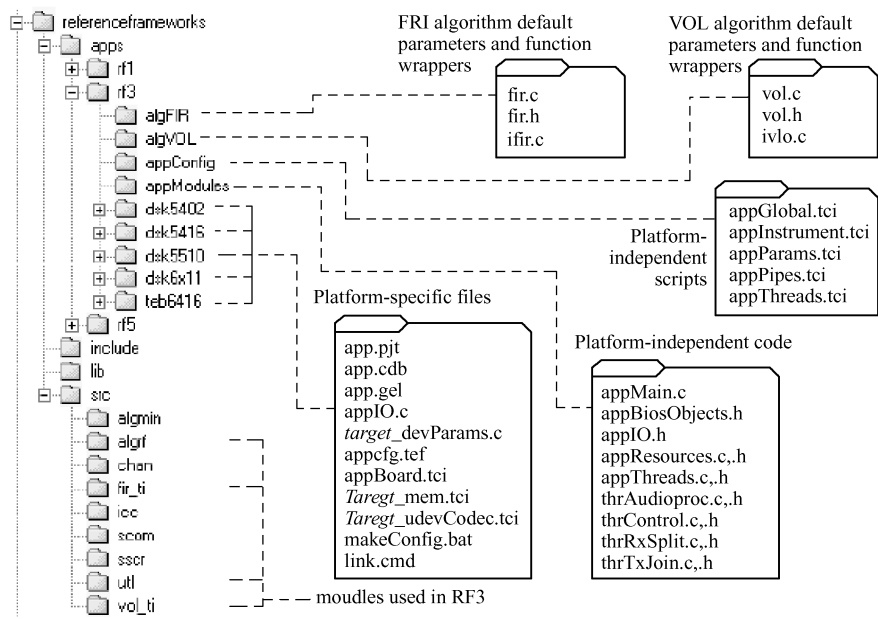


图 2. 20 RF3 文件夹结构

➤ src 包含模块的源文件。在每个文件夹中的 readme. txt 都提供了模块和它们用法的信息。RF3 中使用的模块有：

- algrf，包括 RF3 使用的 IALG 实现的源文件，以初始化 eXpressDSP 算法的 DSP/BIOS MEM 模块或动态内存分配；
- fir\_ti，FIR 滤波器的文件，是 TI 公司提供的用于 RF3 的符合 eXpressDSP 的算法；
- vol\_ti，音量算法的文件，是 TI 公司提供的用于 RF3 的符合 eXpressDSP 的算法；
- utl，包含 UTL 调试诊断模块的源文件。

RF 中没有包括 IOM 设备驱动模块的源代码。那些文件是作为 DSP/BIOS 驱动开发者套件 (DDK) 的一部分。运行 RF 是不需要 DDK 的，因为 RF 包含了驱动库和公共头文件。

RF3 应用程序的模块组织如图 2. 21 所示包括以下几类。



图 2. 21 RF3 模块组织图

➤ 库。库模块一般不需要修改或极少的修改（例如，需要将 IOM 迷你驱动移植到一个不同的设备上）。

- 线程和算法。这部分将在后面章节——配置 RF3 应用中介绍修改方法。
- 初始化。这部分主要涉及到应用程序的 `main()` 函数，它与其他模块调用初始化函数，也可以调用专用的初始化程序。
- 其他。这部分涉及到应用专用的 I/O 函数，例如控制一个设备的 LCD 或 LED 显示，以及闪存的读写等。

#### 2.2.3.4 RF3 应用系统设计方法

在这一节中，我们更准确地描述一个基本 RF3 应用系统，大多数的 RF3 应用能套用这个基本应用，而不用进行实质的修改。这一基本 RF3 应用系统就是前面提到的默认 RF3 示例，它的需求说明书如下。

应用系统获取一个输入音频信号，将它以给定的采样率转换成数字信号；系统将采样值分离，独立地处理各通道。每个音频信号块用两个有符号 16 比特采样值表示，一个给左通道，另一个给右通道。（对于立体声编解码器，信号是分开的。对于单声道编解码器，两个通道是通过将单声道采样值复制左通道和右通道来模拟。）

每个通道分别应用一个滤波器。左通道（第 1 通道）采用低通，右通道（第 2 通道）采用高通。音量控制独立地应用于每个通道，两个通道有可能使用不同的放大/衰减因子。最后，两个通道在输出前合并成单路信号（对于单声道编解码器，只使用一个通道的数据，另一个通道的数据则被丢弃。这个为用户可选的）。产生的结果数据传到输出编解码器。滤波使用 TI 提供的 FIR 算法，音量控制使用 TI 提供的 VOL 算法。

通道的个数是一个可修改的常数，以便系统可以方便地扩展为更多或者更少的通道。

另外，应用系统提供一个控制函数，以便用户可以从主机改变每个通道的音量以及选择哪个通道的数据实际传输到编解码器。用户可以通过 GEL 脚本语言改变参数，来访问目标板存储器，模拟旋钮和开关的值。

基于这个说明书，可以画出这一系统的数据通路如图 2.22 所示。在基本 RF3 应用系统中，贯穿系统的一条数据通路是音频数据从输入编解码器一直到输出编解码器的流动。与大多数 RF3 系统的情形相同，输入同步数据，采样后组成块或帧，并进行处理。一个 RF3 应用可能有超过一条数据流。

使用算法处理一个通道的连续数据块。在立体声系统中，输入数据流有两个通道，输入编解码器分开传输左/右通道的采样值；而单声道编解码器能用来对有  $N$  个编解码器或一个有  $N$  个通道的编解码器的系统进行仿真，例如，对数据采用时分复用（TDM）。

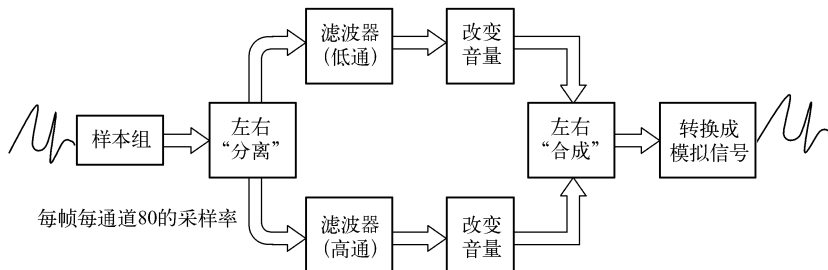


图 2.22 RF3 基本系统的数据通路

在数据通路中，方框表示处理函数，即一个 XDAIS 算法，或一个用户函数或任何它们



的组合。在图 2.22 中, 滤波器和改变音量都是 XDAIS 算法, 而“分离”和“合成”不是 XDAIS 算法。一个方框类似于一个函数, 有零个或更多的输入缓冲区, 产生零个或更多的输出缓冲区和结果的值。

### 1) 在数据通路中确定数据率和缓冲区大小

数据流采用实时处理时, 输出与输入的采样率相同。采样率取决于 RF3 应用程序运行的 DSP 板。RF3 支持在不同的频率运行, 只是如果处理数据的频率越高, CPU 的负荷越大。

图 2.22 中, 每通道每帧 80 个 (16 比特) 采样点, 即一个数据块或帧的大小是 80。在实际系统中, 帧的大小取决于算法需求、系统需求以及可利用的 CPU 时间。默认的 RF3 应用在所有的板子上都使用相同的帧长, 但是用户可以很容易修改帧长。XDAIS 算法一般有固定的帧长, 例如, G. 723 算法一帧有 240 个采样点, G. 729 算法一帧有 80 个采样点。

要改变帧长, 可以在配置中修改 PIP 对象的 framesize 属性, 并且修改文件 appResources.h 中的 FRAMELEN 常数。

当使用 DMA 时, CPU 不干预数据帧的收发工作。于是, 帧长越大, CPU 花在 I/O 上的时间越少, 留给数据处理的时间越多。然而, 较大的缓冲区造成较长的反应时间, 以及占用更多的存储器。因此, 要找到一个折中。

可以通过查看 DSP/BIOS 统计图中的 stsTime0 的值 (设置以毫秒为单位), 看出程序运行时多久处理一次数据缓冲区。这个对象可以测量两个分隔的函数执行之间的时间。测出的值应该大约等于  $1000/\text{采样率} \times \text{帧长}$ ; 当然, 会受到其他 CPU 行为, 如中断处理和周期线程等的影响。

下面介绍一些典型 DSP 目标板的采样率和帧长。

➤ C5402 DSK: C5402 DSK 有一个 8K 采样率的单声道编解码器 (AD50), 每个样点是 16 比特, DSP 处理数据的频率为:

$$1/\text{采样率 (Hz)} \times \text{帧长} = 1/8000 \times 80 = 0.01\text{s} = 10\text{ms}$$

➤ C5510 DSK: C5510 DSK 有一个 44.1K 采样频率的立体声编解码器 (TLV320AIC23), 每个样点是 16 比特, DSP 处理数据的频率为:

$$1/\text{采样率 (Hz)} \times \text{帧长} = 1/44100 \times 80 = 0.00181\text{s} = 1.81\text{ms}$$

对于立体声编解码器, 最开始的输入和最后的输出都是 160 个样点, 每个通道是 80 个。

➤ C6416 TEB: C6416 TEB 有一个 48K 采样频率的立体声编解码器 (PCM3002), 每个样点是 16 比特; 但对于 C6000 系列 DSP, PIP 的帧长是以 32 比特为单位测量的。因此, 包含 40 个 32 比特的 PIP 对象和包含 80 个 16 比特的 PIP 对象帧长相同。DSP 处理数据的频率为:

$$1/\text{采样率 (Hz)} \times \text{帧长} = 1/48000 \times 80 = 1.66\text{ms}$$

对立体声编解码器来说, 最开始的输入和最后的输出都是 80 个 32 比特样点, 而每个通道是 40 个 32 比特样点。

### 2) 在数据通路中确定处理线程

在决定了缓冲区大小和速率之后, 就要进行最重要的设计——把函数组成线程。一个线程在数据通路中由一个或多个函数块组成。函数分组的规则如下:

- 如果两个功能单元并行执行, 它们要放到不同的线程中;
- 如果两个函数执行速率不同, 它们要放到不同的线程中。

当然, 当只有一个单核 DSP 的时候, 不是真正的并行。但是这个并行的宽松理解对我

们的设计目的是足够的。

如果需要, 可以将数据通路分离为更多的线程。实际上, 线程数量的上限是功能单元的数量, 但是把每个功能单元都放入分开的线程是不必要的。

图 2.23 所示为具有三个线程的数据通路, 虚线框表示线程, 一个框中的功能单元属于一个单独的线程。其中, 采样和分组为一个线程; 输出转换为模拟信号作为硬件线程, 控制 DMA 将输出帧的样点送到编解码器; 系统其余所有的模块都以相同的速率运行, 因此可以全部放到一个线程中。

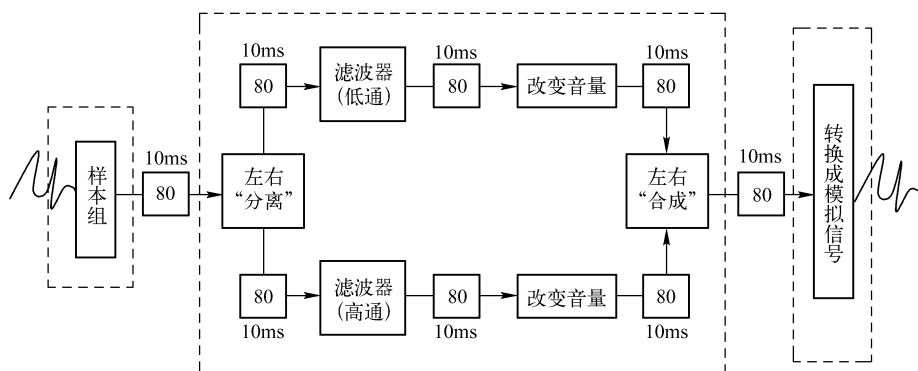


图 2.23 具有三个线程的数据通路

在有立体声编解码器的系统中, 这样的分组是有效的。因为必须左通道和右通道的采样都有时, 才能开始处理输入帧; 而且在两个通道处理完之前, 无法形成输出帧。

然而对于基本应用, 将左通道和右通道分成了不同的线程, 以便示例多通道技术。如果应用不需要多通道, 将所有函数放入一个线程即可。图 2.24 中两个通道分别使用两个线程, 虚线框表示线程。

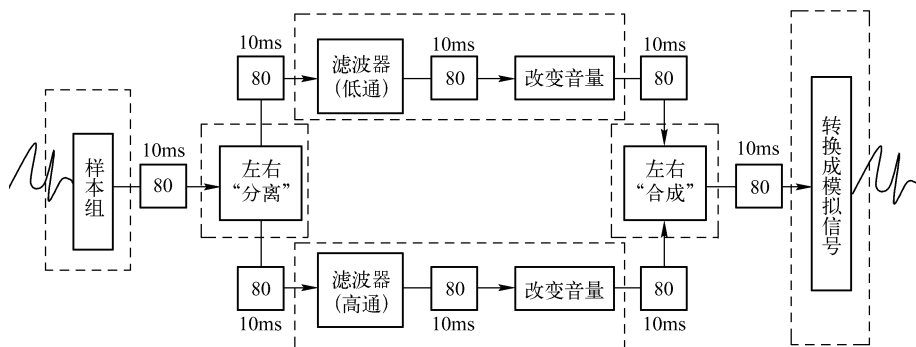


图 2.24 具有六个线程的数据通路

### 3) 使用 DSP/BIOS 模块实现数据通路

由图 2.24 可知, 需要使用 DSP/BIOS 模块实现以下组件:

- 输入/输出线程;
- 数据处理线程;
- 线程间的数据缓冲;
- 线程内的数据缓冲。



线程间的数据缓冲采用 PIP 模块实现。对于线程间通信, 可以用静态队列, 但还必须实现一个同步机制。较好的解决方法是使用提供 DSP/BIOS 的 PIP 模块, 它同时提供了块存储和同步机制。如果线程间需要彻底隔离, 如图 2.24 所示中的输入线程和数据分离线程, 以及数据组合和输出线程, 可以设置 PIP 包括两帧, 从而实现双缓冲或“乒乓”缓冲。

线程内的数据缓冲可以采用简单静态队列来实现。线程处理函数调用滤波算法, 将结果存储到队列, 然后在音量调节算法中使用队列中数据。

输入/输出线程采用 IOM 驱动实现, 包括 PIO 适配器和 IOM 迷你驱动。

IOM 模型把驱动分为两部分: 设备独立层和特定设备层。这种结构使驱动简单和易于维护, 而且把驱动和应用程序分离开, 可以在不同的应用中使用驱动。在 RF3 中, 类驱动和 PIP 对象接口。类驱动是设备独立的, 提供了多线程 I/O 请求的串行性和同步性。类驱动使用设备相关的迷你驱动来操作一个特定设备。与 UNIX 的设备驱动不同, 迷你驱动基本上是一个函数表, 包含若干底层函数, 如“打开设备”、“关闭设备”、“控制设备”、“开始传输”以及“传输完成”的回调函数等。

IOM 驱动传输数据到给定的存储器块中或反之。然而我们使用 PIP 进行线程间的通信, 所以为了简化问题, 可以使用一个 PIO 对象, 即到 IOM 迷你驱动的 PIP 适配器。可以为输入和输出分别配置 PIO 对象, 每个对象使用一个单独的管道。初始化后, 应用程序不用关心硬件, 只需要从输入 PIP 获取数据, 运算后存储数据到输出 PIP 中即可。这样所有数据处理线程可以统一用 SWI 实现, 并且只从 PIP 输入和输出数据。

用 DSP/BIOS 模块实现的数据通路如图 2.25 所示。其中, 一个三角形连接一个表示数据流的箭头代表 IOM 迷你驱动加上一个 PIO 对象; 一个圆柱体代表一个数据管道; 两个连在一起的圆柱体代表“乒乓”缓冲; 一共四个软中断实现的线程。左、右通道分别标记为通道 0 和通道 1。当有多个通道时, 有一个索引会更方便。对象的名字基于它们的类型、它们的函数、它们属于哪一边 (Rx = 接收, Tx = 传送) 以及通道号。

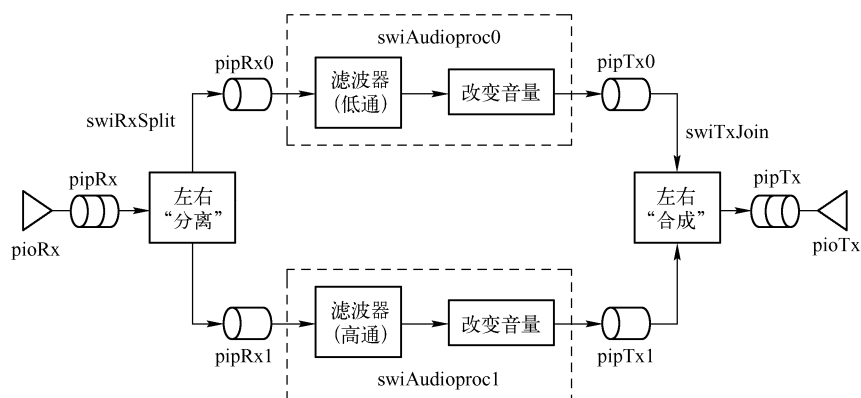


图 2.25 用 DSP/BIOS 实现的数据通路

整个数据通路中数据流总结如下。

- 初始化时, pioRx 被激活。这个函数启动编解码器、McBSP 以及 DMA, 然后立即开始传输第一个输入块。数据放入 pipRx 的第一帧。
- 10ms 后, 80 个采样值到达了, 产生了一个 DMA 中断, 于是调用 pioRx 函数, 这个函数将数据放入 pipRx 的空帧, 并分配新的一帧以及初始化下个块的传输。前面介绍过

不同 DSP 板的采样率和帧长不同, 这里以 C5402 DSK 使用的 8K 采样频率为例。

- 当第一帧已经读入, 就触发 swiRxSplit 线程运行, 将输入数据从 pipRx 复制到 pipRx0 和 pipRx1 中。
- 当数据写入了 pipRx0 和 pipRx1, 就触发 swiAudioproc0 线程和 swiAudioproc1 线程运行。当 swiRxSplit 运行结束时, 先运行 swiAudioproc0、swiAudioproc1 等待。
- swiAudioproc0 线程从 pipRx0 取出数据, 送到 FIR 滤波器, 用中间缓冲区存储滤波结果, 然后运行音量改变算法, 最后将结果存入 pipTx0 管道中。
- 接着运行 swiAudioproc1 线程, 完成相似的功能, 此时使用 pipRx1 和 pipTx1 管道。
- 当 swiTxJoin 线程运行时, 如果 pipTx0 或 pipTx1 中的数据还没准备好, 则选择其中之一的内容复制到 pipTx, 而丢弃其他内容。当 pipTx 写入完成, 就激活 pioTx, 实现从 pipTx 中传输数据到编解码器。
- 输出帧以 10ms (8kHz 的 80 个样值) 准确地传输, 并且产生 DMA 中断。调用 pioTx 检查下一个输出帧是否准备好了, 如果下一个输出帧准备好了就开始传输数据到编解码器。
- 每 10ms 就输入一帧, 上述周期又进行一次, 一直循环往复。系统的处理速率由编解码器/McBSP/DMA 决定, 此例将参数设为 10ms。

#### 4) 加入控制线程

要满足本节开头提到的 RF3 应用系统的需求说明书, 还需要包括控制, 即提供给用户改变每个通道音量和选择通道的方法。

设想这个 RF3 应用系统运行在一个物理设备上, 它有两个音量调节滑块和一个通道选择开关。变更它们的位置表示一个需要的响应硬件事件。我们可以假设这些硬件为 I/O 空间里的寄存器, 有可读的值。要知道一个滑块或开关的值, 应用程序只需要在方便的时间读那些 I/O 寄存器的值, 并把它们应用到处理线程中。

什么时候是读一个外部 I/O 寄存器值的“方便的时间”呢? 这取决于用户改变它的值的时候, 硬件是否对 DSP 芯片发出中断, 如果产生了中断, 则在中断服务 ISR 中读取硬件对应的寄存器。如果改变硬件的值或位置没有导致一个中断, 则只有用轮循环来周期地检查这些值。所以需要有一个周期运行线程, 比如, 每 20ms, 读取 I/O 寄存器, 并更新处理的参数。当然, 周期的选择取决于需要和允许的反应时间。越快地循环检测寄存器, 花费的周期越多, 响应时间也越快。

由于许多板子没有任何可供循环检测的开关, RF3 使用 CCS GEL 语言模拟它们。一个简单的 GEL 脚本可以表示两个滑块和一个开关。只要其中一个改变, 这个脚本就将目标板暂停, 然后在目标板存储器的一个空间中写入改变的值。

如果要执行一个周期的任务, 比如每 20ms 在 DSP/BIOS 里循环检测一次这些“寄存器”, 可以使用 PRD 对象或 CLK 对象。PRD 对象是一个类似 SWI 的软件线程, 它设置为每个用户指定的周期都调用用户指定的程序。应用系统中的时钟周期被设置为每 1ms 发生一次中断。如果使用 CLK 对象, 则在每次定时器中断服务程序中调用一次用户指定的函数。和 PRD 函数不同的是, PRD 函数可以进行冗长的计算, 而 CLK 函数从中断中被调用, 要很简短。

为了举例说明, 我们用一个 CLK 对象模拟 PRD。每 1ms 产生定时器中断, 中断服务程序调用一次 CLK 函数。由于希望每 20ms 循环检测一次“寄存器”, 于是这 20 次中断调用

CLK 函数时, 有 19 次 CLK 函数都立即返回。每 20ms, CLK 函数读控制寄存器, 并将他们的值存储到一个数据结构, 然后启动控制线程 swiControl 运行, 控制线程把这些值应用到数据处理线程中。当获得一个硬件事件中断和需要在硬件值改变时立即读取它们的值, 而不是经过一段延时才去读, 用 CLK 对象的方法是很有效的。但如果不需要这么精确, 可以接受小段延时, 只使用 PRD 就已经足够了。

上述方法将控制步骤分为两部分: 读取那些值和应用它们。本例中, 当控制线程 swiControl 运行时, 这些值才被应用, 控制线程改变音量控制算法的两个实例中的参数, 以及 TxJoin 线程中的值, 以决定让哪个信道通过, 哪个不通过。

控制线程什么时候运行取决于它的优先级。控制线程的优先级不应比处理线程的优先级高, 否则, 处理线程会被控制线程抢占。当处理线程恢复运行时, 它的参数改变了, 于是接着用新的参数处理, 这很可能产生错误的计算结果。如果给控制线程比处理线程更低的优先级, 处理线程就不会有计算的连贯性问题了。但是控制线程就可能一直得不到运行机会, 导致参数改变毫无反应。

一个较好的解决方案是让控制线程和处理线程具有相同的优先级。这样既没有连贯性问题也没有无反应的情况。如果处理线程在控制线程发出参数改变信号时正在运行, 控制线程会等待处理线程完成后才改变参数值; 如果控制线程发出参数改变信号时处理线程没有运行, 则控制线程会立即改变参数值。

改变线程参数通常是通过调用 XDAIS 算法的 control() 函数完成的, 但可以是任何影响数据处理线程的函数。RF3 系统基于 DSP/BIOS 的数据通路加上控制线程的最终框图如图 2.26 所示。

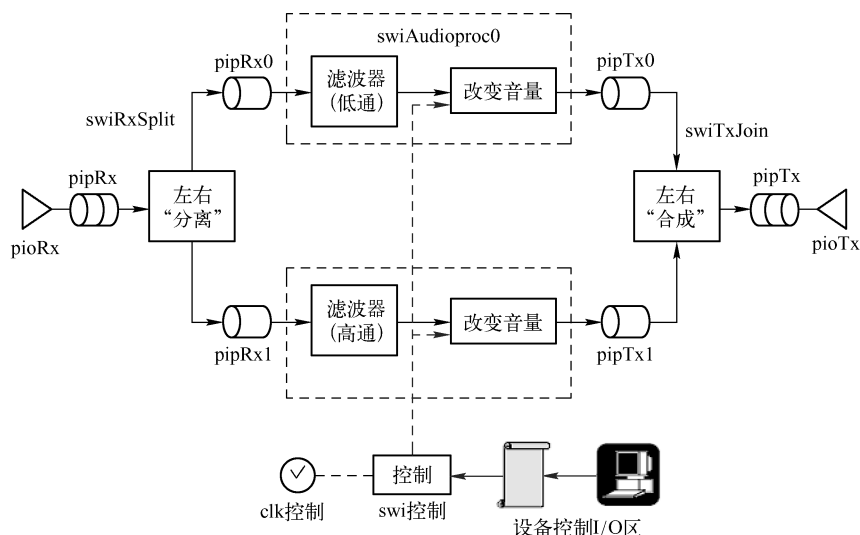


图 2.26 RF3 系统最终的数据通路

### 2.2.3.5 RF3 源码解析

在 2.2.3.3 节已经介绍了 RF3 文件夹结构, 本节对源代码结构和部分代码进行解析。

图 2.27 显示了 RF3 头文件的编译流程。图中箭头尾部的文件包含了箭头开端的文件。

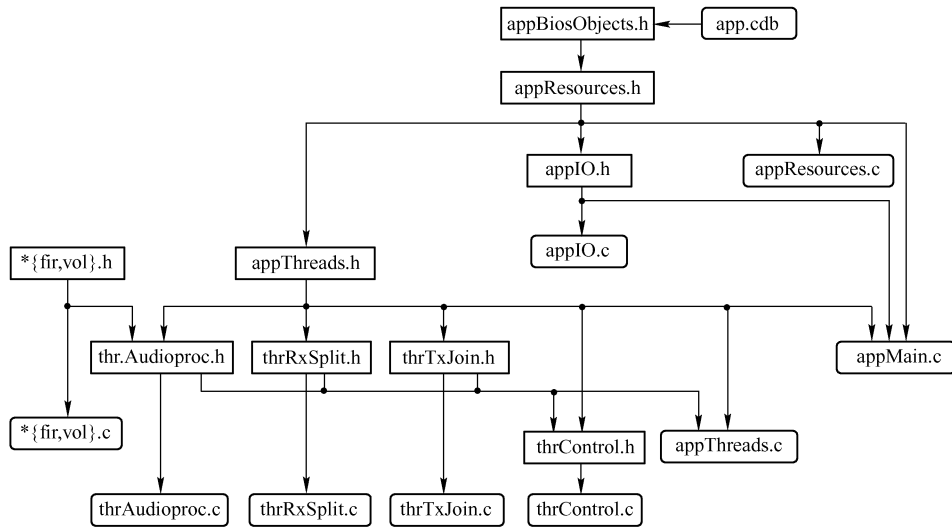


图 2.27 RF3 头文件编译流程

RF3 应用中的非 DSP/BIOS 全局数据对象有：

1) Audioproc 线程的数据结构如下。

```

typedef struct ThrAudioproc {
    FIR_Handle algFIR; /* an instance of the FIR algorithm */
    VOL_Handle algVOL; /* an instance of the VOL algorithm */
    PIP_Handle pipIn; /* input pipe(s) */
    PIP_Handle pipOut; /* output pipe(s) */
    Sample * bufInterm; /* intermediate buffer(s) */
    /* everything else that is private for a thread comes here */
} ThrAudioproc;

extern ThrAudioproc thrAudioproc[ NUMCHANNELS ];

```

2) RxSplit 线程的私有结构以及初始化。

```

typedef struct ThrRxSplit {
    PIP_Handle pipIn; /* input pipe with joined channel data */
    PIP_Handle pipOut[ NUMCHANNELS ]; /* outpipes with individ. channel data */
} ThrRxSplit;

ThrRxSplit thrRxSplit = {
    &pipRx, /* pipIn */
    { &pipRx0, &pipRx1 } /* pipOut[ NUMCHANNELS ] */
};

```

RF3 应用的函数调用层次如图 2.28 所示。

下面重点讨论 RF3 源代码中需要修改的地方，以便以此为基础建立自己的应用程序。

1) appBiosObjects.h

包括了应用程序中要用到的所有 DSP/BIOS 对象的声明，包括 PIP 对象和 LOG 对象。这

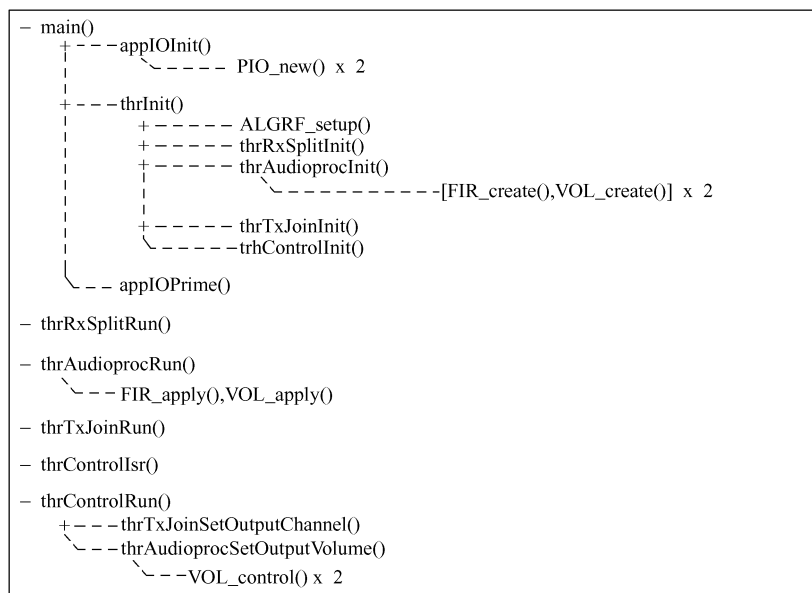


图 2.28 RF3 函数调用层次

些对象对每个模块都可见。只要对配置更改名称或添加新的对象，都需要更新这个文件。可以通过图形化的方式配置 BIOS 对象，就会自动地生成这个头文件。

## 2) appResources. h

包括了应用程序中使用的宏、数据类型、函数声明等。在 RF3 例子中，这个文件定义了通道数 NUMCHANNELS、帧的大小 FRAMELEN，以及一个数据样点的类型。由于 short 在 'C5000 和 'C6000 DSP 上都是 16 比特，而管道是以字（word）为单位计算大小的，在 'C5000 上一个字是 16 比特，但是在 'C6000 上一个字是 32 比特。为了避免混淆，appResources. h 文件中定义了两个宏 sizeInSamples 和 sizeInWords，分别说明了样点的比特数和字的比特数。

## 3) appResources. c

本文件应该定义在 appResources. h 中声明的所有全局变量，也就是为它们分配空间和初始化。RF3 例子中没有任何这样的变量（除了用于调试的 UTL\_sts 对象），所以这个文件是空的。

## 4) appIO. h

appIO. h 文件声明了两个 I/O 函数，appIOInit() 和 appIOPrime()，用于初始化 I/O 设备和启动 I/O 管道。启动指把初始的零帧放到管道中。启动必须是一个和初始化分开的操作，因为在 main() 中，通常 I/O 设备首先被初始化然后才能启动。

这个文件还声明了两个 PIO 对象 pioRx 和 pioTx，分别对应输入和输出端。这些对象在 appIO. c 中使用，假设需要在其他地方访问它们，就要把它们设为全局可见。

## 5) appThreads. h

和 appResources. h 类似，本文件定义的信息贯穿了整个应用程序。appThreads. h 定义宏和常量，声明变量和线程中用到的函数。

RF3 例子中的这个文件声明了几个 Audioproc 线程使用的中间缓冲区；还声明了 thrInit() 函数，此函数初始化了所有的线程，被 main() 函数调用。

## 6) appMain. c

这个文件为 `main()` 函数, 包含了 `appResources. h`、`appIO. h` 和 `appThreads. h`。`main()` 函数调用 `appIOInit()`、`appThreadInit()` 和 `appIOPrime()` 等, 执行必要的初始化操作。当 `main` 函数退出则将控制权交给 DSP/BIOS。

## 7) appIO. c

`appIO. c` 文件是与特定目标板相关代码的核心部分。在 RF3 文件夹中为每个目标板都提供了不同的实现版本。本文件将不同的编解码器和采样率的调用封装起来, 定义了两个 PIO 对象, 实现 I/O 的初始化和启动。

为了初始化 I/O, `appIO. c` 文件为每个 PIO 对象调用 `PIO_new()`。`PIO_new()` 的调用包括 PIO 对象的地址、相关联的管道地址、IOM 迷你驱动的名称、操作模式 (输入或输出), 以及驱动函数表的地址。

启动函数放置两个空帧到 `pipTx` 中, 以便确保有连续的输出并避免听到开始的砰砰和咔哒声。启动时放置两个空帧可以起到填充空隙的作用, 实现持续的输出。

为了解释启动的作用, 让我们看一个例子。假设一共有 80 个以 8K 采样率采样的 16 比特样本值。即每 10ms 会到来一个新的输入帧。显然, 每 10ms 也必须产生一个新的输出帧。换句话说, 一个输出帧传输完成后, 下一个处理过的帧就应该准备好了。如果不是这样, 输出就会有一段空闲直到下个输出帧准备好, 不能得到持续的输出, 这并不是我们想要的。

如图 2.29 所示, 假定第一帧和第二个帧花了 4ms 来处理, 而第三帧因为某些原因需要 7ms 来处理, 输入和输出 pipe 都是双缓冲。第一帧在  $t = 14\text{ms}$  时输出, 第二帧在  $t = 24\text{ms}$  时完成处理, 正好是第一帧输出传输完成的时间。然而, 在  $t = 34\text{ms}$ , DMA 已经准备好传输第三个输出帧, 但第三帧还没处理完。当  $t = 37\text{ms}$  时, 第三帧才准备好, 这时才可以开始传输第三个输出帧。于是产生了 3ms 的间隙。所以即使算法正确地在 10ms 内完成处理, 也会错过实时传输的截止时间, 产生错误的输出。

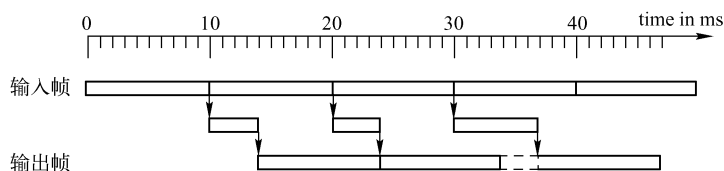


图 2.29 输出管道没有填充启动帧

通过启动时填充两个空帧到输出管道可以轻松地解决这个问题。如图 2.30 所示, 两个初始的输出帧全是零, 后面就可以得到连续的输出。即使第三个帧要花 7ms 来处理, 但也能满足实时传输的截止时间。只要处理时间在 10ms 以内, 就不会影响到输出的连续性, 因为输出帧早就准备好了。

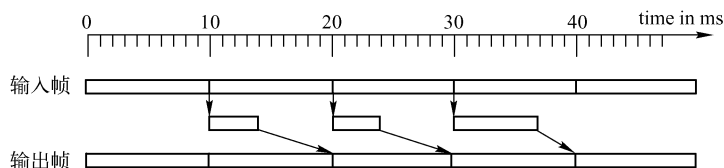


图 2.30 输出管道填充了启动帧



## 8) thrRxSplit. h

此头文件声明 RxSplit 线程的数据对象 thrRxSplit。RxSplit 线程很简单，只需要让句柄指向输入管道和两个输出管道。

这个文件还声明了 RxSplit 线程的两个公共函数：thrRxSplitInit() 和 thrRxSplitRun()。前者由 main() 函数调用。后者在 SWI 中自动调用，不是直接由应用程序的代码调用。

## 9) thrRxSplit. c

这个文件执行 RxSplit 线程。像其他线程一样，包括三个部分。

- 线程数据对象的静态初始化。包括将线程的管道句柄连接到实际管道的地址，如果我们将线程放入数据通路的其他地方，我们只需要改变这个连接。
- 线程运行时的初始化。由 thrRxSplitInit() 函数完成，它是空函数，因为 RxSplit 线程很简单。
- 被 SWI 对象调用的 run() 函数。thrRxSplitRun() 函数被 swiRxSplit 对象调用。当它的邮箱从二进制 111 减至 000 时，swiRxSplit 自动被激活。此 SWI 使用的三个管道分别通过调用 SWI\_andnHook() 函数将线程邮箱中的一个比特清掉；所以当线程被激活时，它“知道”输入管道中有一帧数据可以读入，并且两个输出管道都分别有一个空闲帧可以写入。

通过 PIP\_get() 函数，使用 pipIn 句柄而不是 pipRx 的名称获取输入管道；通过调用 PIP\_getReaderAddr() 和 PIP\_getReaderSize() 确定输入帧的地址和大小。然后在循环中，分配每个输出管道，并将空闲帧的地址存入一个数组（数组大小为 NUMCHANNELS）。

thrRxSplitRun() 函数的主要工作是将一个源帧复制到每个目的帧。对于一个立体声编解码器，需要复制每个源采样点到一个目的地。对于一个单通道编解码器来说，我们复制所有的采样点。最后，我们释放输入管道，函数就退出了。

## 10) thrTxJoin. h

这个文件类似于 thrRxSplit. h 文件，有一个小的区别是 ThrTxJoin 结构中有一个 Int outputChannel 成员，来存放传送输出帧的通道数量。控制线程可以基于用户的行为更改它。

## 11) thrTxJoin. c

这个文件也类似于 RxSplit，不同的是它根据 thrTxJoin.outputChannel 的值，决定复制哪一个输入到仅有的输出，然后丢弃另一个。

需要注意，即使对于不使用的那个通道，也必须获取和释放输入管道，才能让系统工作。否则，管道就不会调用它的 notifyWriter 函数，相应 swiAudioproc (0 或 1) 线程的邮箱不会被清除，线程就不会被激活，处理线程就不会读取它的输入管道 (pipRx0 或 pipRx1)，那个管道也不会调用它的 notifyWriter 函数，于是 swiRxSplit 线程不会被激活，然后整个系统在两帧后就无法工作了。

## 12) fir. h、fir. c、vol. h、vol. c

这四个文件为两个模拟 XDAIS 算法提供函数封装。让应用程序以一种“友好的”方式调用算法函数。比如，应用程序用 FIR\_apply(<参数>) 代替了 firHandle->fxns->filter(<params>) 的复杂语法。这些函数封装提高了代码的可读性。

通常，alg 是算法的名字，需要实现 alg\_create()、alg\_delete()、alg\_activate()、alg\_deactivate()、alg\_init()、alg\_exit()、alg\_control() 以及特定算法函数，比如 FIR 算法的 FIR\_apply。

一般由算法提供商提供封装文件。为了 RF3 中使用这些封装文件，需要做一些修改，

即用 ALGRF 模块代替大多数封装默认使用的 ALG 模块。

### 13) ifir.c、ivol.c

文件 ifir.c、ivol.c 包含默认的算法实例参数，由算法提供商提供。只需要复制这些文件到工程文件夹中，并把它们包含进来。除了在初始化时加载参数值，一般不要更改这些文件。

### 14) thrAudioproc.h

thrAudioproc.h 定义了 Audioproc 线程的数据结构和外部接口。前面已经给出了这个数据结构。Audioproc 线程有一个 FIR 实例的句柄、一个 VOL 实例的句柄、一个输入 PIP 句柄、一个输出 PIP 句柄，以及一个指向中间缓冲区的指针。通过将私有数据分组，创建线程的队列，每个通道一个 Audioproc 线程。当一个线程运行时，可以知道它的索引——通道号，以便访问它的数据结构。

### 15) thrAudioproc.c

这个文件是应用的核心，通过调用 XDAIS 算法执行处理数据流的线程。包括三个部分。

- 线程数据对象的静态初始化。声明线程对象并将指针连接到数据通路中的实际对象。因为 Audioproc 是一个输入管道、一个输出管道以及一个中间缓冲区。虽然我们尽量静态初始化，但 ALGRF 模块必须动态创建 XDAIS 算法实例，所以我们使用空指针作为队列中算法线程指针的初始值。
- 线程运行时的初始化。接下来的阶段是动态初始化。代码首先准备动态初始化要使用的静态参数。在默认情况下，有两组 FIR 滤波器系数，一组是低通滤波器，另一组是高通滤波器。thrAudioprocInit() 函数必须初始化所有 Audioproc 线程的实例，在我们的例子中，包括 thrAudioproc[0] 和 thrAudioproc[1]。对于每个线程，在一个循环中创建和初始化所有使用的 XDAIS 算法实例。即首先创建一个 FIR 实例和一个 VOL 实例，然后初始化线程的数据结构中的 algFIR 域和 algVOL 域。通过对本地参数变量赋以缺省的参数创建 FIR 实例，对 VOL 实例也执行类似操作。
- 被 SWI 对象调用的 run() 函数。thrAudioprocRun() 函数被 swiAudioproc0 和 swiAudioproc1 对象调用，分别传递 0 和 1 作为参数。通道参数是通道号，用于访问正确的线程数据结构。这个函数按常规模式获取输入管道和分配输出管道，确定帧的地址和大小，处理输入帧，调用 FIR 和 VOL 后存储结果到输出帧。FIR 的输入是输入帧，输出是中间缓冲区。VOL 的输入是中间缓冲区，输出是输出帧。最后函数释放输入管道并且退出前发送输出。

下面为 thrAudioprocRun() 函数的代码，省略了 UTL 调用。

```
Void thrAudioprocRun( Arg aChan)
{
    Sample *src, *dst;
    Int size; /* in samples */
    Int chan;
    chan = ArgToInt( aChan );
    /* get the full buffer from the input pipe */
    PIP_get( thrAudioproc[ chan ]. pipIn );
```

```

src = PIP_getReaderAddr( thrAudioproc[ chan ]. pipIn );
/* get the size in samples (the function below returns it in words) */
size = sizeInSamples( PIP_getReaderSize( thrAudioproc[ chan ]. pipIn ) );
/* get the empty buffer from the out - pipe */
PIP_alloc( thrAudioproc[ chan ]. pipOut );
dst = PIP_getWriterAddr( thrAudioproc[ chan ]. pipOut );
/* apply filter and store result in intermediate buffer */
FIR_apply( thrAudioproc[ chan ]. algFIR,src,thrAudioproc[ chan ]. bufInterm );
/* amplify the signal in the interm. buffer and store result in dst */
VOL_apply( thrAudioproc[ chan ]. algVOL,thrAudioproc[ chan ]. bufInterm,dst );
/* Record the amount of actual data being sent */
PIP_setWriterSize( thrAudioproc[ chan ]. pipOut,sizeInWords( size ) );
/* Free the receive buffer,put the transmit buffer */
PIP_free( thrAudioproc[ chan ]. pipIn );
PIP_put ( thrAudioproc[ chan ]. pipOut );
}

```

另外, thrAudioproc. c 文件还包括 thrAudioprocSetOutputVolume() 函数, 被控制线程调用来设置输出音量。它从 thrAudioproc 结构中获取 VOL 实例的句柄, 然后调用 control() 函数改变音量。注意我们改变 VOL 实例的状态是通过首先读取现有的状态, 接着在返回的结构中更改一个单独的域, 然后传递新的结构到算法的 control() 控制线程。这样确保了我们没有在本地创建的参数结构中传递未初始化的值。

#### 16) thrControl. h、thrControl. c

控制线程是一个 PRD 线程。其中的一部分是 thrControlIsr() 函数, 它在每次“滴答”产生时钟中断时被 CLK 对象 clkControl 调用。这个中断服务程序读取模拟的 I/O 寄存器的值, 其中包括两个滑块值 (每个滑块指示对应通道的音量) 和一个开关值 (用于选择活动通道)。这些在“deviceControlsIOArea”整数队列里的 I/O 寄存器的值实际上是由 GEL 脚本或浏览窗口中的调试器写入的。每到二十次“滴答”, thrControlIsr() 不立即返回, 它读取 I/O 寄存器的值, 并解释这些值, 将结果 (通道音量和活动通道号) 存储在控制线程的数据结构 thrControl 中。

控制线程和数据处理线程优先级相同。swiControl 对象调用 thrControlRun(), 在得到输出通道的本地拷贝时关闭硬件中断, 并从 thrControl 中获取音量变量的值; 然后重新开中断, 在 TxJoin 线程的数据结构 thrTxJoin 中设置 outputChannel 的值; 最后, 为每个通道调用一次 thrAudioprocSetOutputVolume() 函数 (在文件 thrAudioproc. c 中)。

#### 17) appThreads. c

appThreads. c 的目的是定义所有线程使用的全局变量, 我们的例子中是定义中间缓冲区 bufAudioproc[ FRAMELEN ], 以及实现所有线程的初始化函数 thrInit()。

thrInit() 函数首先调用 ALGRF\_setup() 分配内部和外部的堆。当 XDAIS 算法请求存储器时, 可能会要求不同类型的存储器, 实际上就分为内部的或外部的存储器。内部的堆在内部存储器段中创建, 有一个 INTERNALHEAP 标识符。外部堆的标识符是 EXTERNALHEAP。如果某系统中只有内部存储器, 可以调用 ALGRF\_setup (INTERNALHEAP、INTERNAL-HEAP), 这样迫使“外部的”段也在内部存储器分配。接着为所有线程调用 init() 函数, 然

后返回。注意我们不返回初始化是否成功的信息，但是我们通过 `UTL_assert()` 调用知道任何故障，`UTL_assert()` 是调试模块的一部分，如果初始化失败它会中止应用。

#### 18) link.cmd

连接命令文件 `link.cmd` 管理连接过程。它的第一步是，包含自动生成的 `appcfg.cmd` 文件，这个文件把不同的执行段放置到恰当的存储器区域，由 DSP/BIOS 的 MEM 对象定义的。`appcfg.cmd` 文件还生成多种其他的 DSP/BIOS 连接信息。

在我们的应用中，`link.cmd` 文件包含所有的库：包括 IOM 迷你驱动、PIO、UTL、ALGRF、FIR\_TI 以及 VOL\_TI 模块和算法的库。

#### 19) app.gel

GEL 源文件很简单。它显示三个滑块，其中一个可以选 0 或 1，用作两个通道的选择器；还有两个音量滑块，每个通道一个。每次用户移动一个滑块，对应的 GEL 程序就被调用，向目标的 `deviceControlsIOArea` 队列写入新值。当写入时，CCS 临时中止目标板和存储这些值，然后再恢复。这个暂停会导致输出声音轻微的中断。

### 2.2.3.6 基于 RF3 的算法实现例

这一节讨论如何基于 RF3 建造自己的应用系统。参考框架设计为尽可能达到不需要修改主要的源文件和连接文件，就可以改变算法和通道。

从 RF3 创建一个用户应用系统，首先需要为自己的硬件写新的 I/O 驱动；其次，数据通路可能会与 RF3 中的不同；而且一些应用相关的和硬件相关的 I/O 需要集成到数据处理线程中。

下面介绍基于 RF3 建造自己的应用系统的例子，采用单通道，并使用 G.726 编解码 XDAIS 算法替代 FIR 和 VOL 算法。硬件平台和 I/O 驱动不变。修改步骤如下：

- 创建应用文件夹；
- 移除第二个通道；
- 移除控制线程；
- 将库和头文件放到工程文件夹中；
- 创建算法封装；
- 修改代码以使用新的算法；
- 修改工程。

由于此例子仅编码并立即解码音频数据，不具有实际应用价值。但可以说明 RF3 中加入/更新一个 XDAIS 算法的步骤。SWI 线程仍然使用一个中间缓冲区，它存储数据编码的结果作为解码块的输入。新的数据通路如图 2.31 所示。

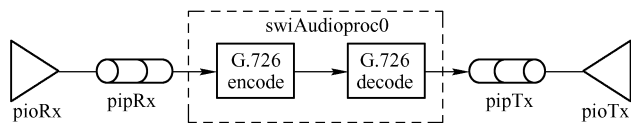


图 2.31 单通道、无控制线程和使用 G.726 算法的数据通路

#### 1) 创建应用文件夹

在 `apps\` 文件夹中，复制整个 `rf3\` 文件夹树，然后命名为 `rf3_vocode`。

下面的文件路径中，`TI_DIR` 是指 CCS 安装的默认路径 `c:\ti`，而 `RF_DIR` 是指参考框架文件夹树的顶层。其他文件路径和刚才创建的文件夹位置相关（例如，`RF_DIR\apps\rf3_vocode`）。

在 CCS 中, 打开工程文件 RF\_DIR\apps\rf3\_vocode\target\app. pj (例如, RF\_DIR\apps\rf3\_vocode\dsk5402\app. pj), target 为所支持的目标板。

## 2) 移除第二个通道

首先修改配置: 打开 DSP/BIOS 配置文件 app. cdb。

删除表示第二个通道处理、输入分离线程和输出接合线程的 swiAudioproc1、swiRxSplit 和 swiTxJoin 对象, 删除 pipRx0、pipRx1、pipTx0 和 pipTx1 管道。现在 Audioproc0 是唯一的数据处理线程, 直接读和写数据到/从 PIO 对象, 使用 pipRx 和 pipTx。

右键点击 pipRx, 然后选择属性。更改 notifyReader 函数的参数 nrarg0 为 \_swiAudioproc0, 使 notifyReader 函数通过调用 SWI\_andnHook (swiAudioproc, 1) 去清除 swiAudioproc0 邮箱中的第一个比特。右键点击 pipTx, 然后选择属性。更改 notifyWriter 函数的参数 nwarg0 为 \_swiAudioproc0。修改参数 nwarg1 为 2 (二进制 010), 以便在管道准备好时, notifyWriter 函数调用 SWI\_andnHook (swiAudioproc, 2) 来清除第二个比特。

保存更改后的 app. cdb 文件。

然后要修改和第二通道相关的代码。

在工程中移除不再需要的 thrRxSplit. c 和 thrTxJoin. c 文件。

打开 appModules\appBiosObjects. h 文件, 移除下列 PIP 对象的声明。

```
extern PIP_Obj pipRx0;      /* receive pipe - channel 0 */
extern PIP_Obj pipTx0;      /* transmit pipe - channel 0 */
extern PIP_Obj pipRx1;      /* receive pipe - channel 1 */
extern PIP_Obj pipTx1;      /* transmit pipe - channel 1 */
```

打开 appModules\appResources. h 文件, 将通道数宏 NUMCHANNELS 的定义变为 1, 并保存。

打开 thrAudioproc. c 文件, 进行如下修改。

- 在 thrAudioproc[ NUMCHANNELS ] 队列声明中的 channel #0 部分, 用 &pipRx 替代 &pipRx0 以及用 &pipTx 替代 &pipTx0。
- 从 thrAudioproc[ NUMCHANNELS ] 队列声明中移除掉整个 channel #1 的部分。
- 从 filterCoefficients[ NUMCHANNELS ][ 32 ] 初始化中去掉大括号里的两组滤波器系数的其中一组。
- 由于去掉了 TxJoin 线程, 我们只有在 thrAudioprocRun() 中清除掉每个输出采样的最后一个比特了。所以, 在 thrAudioprocRun() 函数的开头添加声明 int i; 在调用 VOL\_apply() 后, 添加下面的语句。

```
/* For mono codecs, clear last bit in each sample from destination buffer */
#ifdef APPMONOCODEC
for (i = 0; i < size; i++) {
    dst[i] = dst[i] & 0xfffe;
}
#endif
```

打开 thrControl. c 文件, 进行如下修改。

- 去掉 #include " thrTxJoin. h "
- 修改 deviceControlsIOArea 的初始化, 移除第二个通道 channel #1 默认的音量值 100。



- 在 thrControlRun() 函数中, 移除 outputChannel 变量的声明和赋值。
- 在 thrControlRun() 函数中, 移除告诉 thrTxJoin 线程使用哪个通道来输出的语句, 因为我们不再使用 TxJoin 线程了。

```
/* Instruct thrTxJoin thread which channel to output */
thrTxJoinSetOutputChannel(outputChannel);
```

打开 appThreads.c 文件, 进行如下修改。

- 移除包含 thrRxSplit.h 和 thrTxJoin.h 文件的行。

```
#include "thrRxSplit.h" /* definition of the multiplexing thread */
#include "thrTxJoin.h" /* definition of the demultiplexing thread */
```

- 从 thrInit() 程序中移除调用 thrRxSplitInit() 和 thrTxJoinInit() 函数的行。

保存所有编辑过的源文件, 并保存工程。然后重新 build。

到这一步, 已经是一个完整的修改示例, 成为了一个单通道的工程。下面将说明如何修改工程以使用不同的编解码算法。

### 3) 移除控制线程

打开 DSP/BIOS 配置文件 app.cdb。删除 SWI 对象 swiControl; 删除 CLK 对象 clkControl; 保存修改后的 app.cdb 文件。

从工程中移除 thrControl.c 文件。也可以从 appModules\文件夹中移除 thrControl.\* 文件。

打开 appModules\appThreads.c 文件, 进行如下修改。

- 去掉#include "thrControl.h" 这行。
- thrInit() 程序中去除调用 thrControlInit() 函数的行。

打开 appModules\appBiosObjects.h 文件, 去掉 SWI 对象的外部声明。

```
/* SWI objects: declare those that are or might be posted manually */
extern SWI_Obj swiControl; /* control thread */
```

### 4) 将库和头文件放到工程文件夹中

和任何 XDAIS 算法一样, G.726 编解码器包含四部分: 库、头文件、函数封装和默认参数。为了简化这些文件的路径, 我们把库放在公共库文件夹中, 头文件放在公共头文件夹中, 封装和参数放在本地文件夹里。

TI 提供的 G.726 编解码例子算法的文件在 TI\_DIR\examples\target\xdaishdemo\lib 和 TI\_DIR\cXXXX\xdaishsrc\vocoders 文件夹中。如果使用其他算法, 需要参阅算法提供商的文档。

首先复制库文件, 从 TI\_DIR\examples\target\xdaishdemo\lib 复制到 RF\_DIR\lib。

然后复制接口头文件。从 TI\_DIR\cXXXX\xdaishsrc\vocoders 复制到 RF\_DIR\include。

### 5) 创建算法封装

算法封装让应用以一种友好的方式调用算法函数。算法提供商一般都提供了封装文件。大多数算法封装以.h 文件中的静态内联函数形式实现。

大多数默认的算法封装中使用 ALG 模块, 而 RF3 中 XDAIS 算法实例使用 ALGRF, 所以需要修改。如果已经提供了封装文件, 可以按以下步骤修改。

- 在 apps\rf3\_vocode\文件夹中, 创建 algG726ENC\和 algG726DEC\文件夹。用来放函



数封装和默认参数文件。

- 对于编码器，将源文件和头文件从 TI\_DIR\cXXXX\xdais\src\vocoders 复制到 RF\_DIR\apps\rf3\_vocode\algG726ENC。
- 对于解码器，将源文件和头文件从 TI\_DIR\cXXXX\xdais\src\vocoders 复制到 RF\_DIR\apps\rf3\_vocode\algG726DEC。
- 对下面复制的封装文件，做下列区分大小写的全局查找和替换：把 ALG 改为 ALGRF。注意不要修改 IALG。还需要把 alg.h 改为 algrf.h。

```
apps\rf3_vocode\algG726ENC\g726enc.h
apps\rf3_vocode\algG726ENC\g726enc.c
apps\rf3_vocode\algG726DEC\g726dec.h
apps\rf3_vocode\algG726DEC\g726dec.c
```

如果没有封装文件，则按以下步骤修改（myalg 是算法名的小写，MYALG 是大写）。

- 在 apps\rf3\_vocode\文件夹中，创建一个 algMYALG\文件夹。
- 复制 algVOL\vol.h 和 algVOL\vol.c 到 algMYALG\中，然后重命名 myalg.h 和 myalg.c。
- 编辑 myalg.h 和 myalg.c，用“MYALG”替代每个“VOL”实例，用“myalg”替代每个“vol”实例。

#### 6) 修改变代码以使用新的算法

所有引用 FIR 和 VOL 算法的代码都放在 Audioproc 线程文件中，所以只有文件 thrAudioProc.h 和 thrAudioProc.c 需要修改。

**注意：**我们在这里将所有调用 FIR 替换成 G726ENC，VOL 替换成 G726DEC。可是，FIR 和编码器之间没有任何关联，VOL 和解码器之间也没有任何关联。这种替换仅仅因为原来的例子是先调用 FIR 后调用 VOL，这个顺序和先编码后解码的顺序一致。

首先打开 thrAudioProc.h 文件，进行区分大小写的全局修改：将 FIR 换成 G726ENC、fir 换成 g726enc、VOL 换成 G726DEC、vol 换成 g726dec。这些修改包括#include 算法句柄声明，以及算法函数结构声明。然后去掉下面的 thrAudioProcSetOutputVolume() 函数声明。

```
/* parameter change function, called by the control thread */
extern Void thrAudioProcSetOutputVolume( Int chan, Int volume );
```

打开 thrAudioProc.c 文件，进行区分大小写的全局修改：将 FIR 换成 G726ENC、fir 换成 g726enc、VOL 换成 G726DEC、vol 换成 g726dec。

在 thrAudioProc.c 中，去掉下面的静态 filterCoefficients 数组声明和初始化语句。

```
static Sample filterCoefficients[ NUMCHANNELS ][ 32 ] = {
/* 8kHz, 32 Taps, 800 Hz, 1300Hz, pass ripple 0.897dB, stop atten 51 dB */
{
0x0015, 0x0043, 0x0085, 0x00AD, 0x006C, 0xFF69, 0xFD7D, 0xFAEC,
0xF88C, 0xF79B, 0xF957, 0xFE63, 0x0654, 0x0F94, 0x17C7, 0x1C9A,
0x1C9A, 0x17C7, 0x0F94, 0x0654, 0xFE63, 0xF957, 0xF79B, 0xF88C,
0xFAEC, 0xFD7D, 0xFF69, 0x006C, 0x00AD, 0x0085, 0x0043, 0x0015,
```

```

},
};

```

在 thrAudioproc. c 中, 修改编码器的默认参数。去掉下面划掉的行, 添加粗体的行。

```

/* Set the parameters structure to the default, i. e.
 * the one used in i < alg >. c, and modify fields that are different.
 */
g726encParams = G726ENC_PARAMS;          /* default parameters */
g726encParam. cecffptr /* filter coefficients */
(Short *)filterCoefficients[i];
g726encParam. filterLen /* filter size */
sizeof(filterCoefficients[i]) /* sizeof(Sample);
g726encParams. frameLen = FRAMELEN;        /* frame size in samples */
g726encParams. mode      = IG726_LINEAR;

```

在 thrAudioproc. c 中, 修改解码器的默认参数。去掉下面划掉的行, 添加粗体的行。

```

/* do the same for the G726DEC algorithm; create parameters structure */
g726decParams = G726DEC_PARAMS;            /* default parameters */
g726decParam. frameSize = FRAMELEN; /* size in samples */
g726decParam. gainPercentage = 100; /* default gain */
g726decParams. frameLen = FRAMELEN;
g726decParams. mode      = IG726_LINEAR;

```

在 G726ENC\_apply() 调用时, 将第二参数强制转换成 (XDAS\_Int16 \*), 将第三参数强制转换成 (XDAS\_Int8 \*)。调用语句如下:

```

G726ENC_apply( thrAudioproc[ chan ]. algG726ENC, (XDAS_Int16 *)src, (XDAS_Int8 *)
thrAudioproc[ chan ]. bufInterm );

```

在 G726DEC\_apply() 调用时, 将第二参数强制转换成 (XDAS\_Int8 \*), 将第三参数强制转换成 (XDAS\_Int16 \*)。调用语句如下:

```

G726DEC_apply( thrAudioproc[ chan ]. algG726DEC, (XDAS_Int8 *)thrAudioproc[ chan ].
bufInterm, (XDAS_Int16 *)dst );

```

从 thrAudioproc. c 中去掉整个 thrAudioprocSetOutputVolume() 函数。

修改注释, 保存所有文件。

#### 7) 修改工程

从工程中移除 fir. c、ifir. c、vol. c 和 ivol. c。

在工程中加入下列参数文件和算法封装文件。

algG726ENC\ig726enc. c
algG726ENC\g726enc. c
algG726DNC\ig726enc. c
algG726DNC\g726enc. c

在工程窗口，打开 link.cmd 文件，进行如下修改。

查 找	替 换
- Ifir_ti. IXX ( where XX is the two - digit platform)	- Ig726enc_ti. IXX
_FIR_IFIR = _FIR_TI_IFIR;	_G726ENC_IG726ENC = _G726ENC_TI_IG726ENC;
- Ivol_ti. IXX	- Ig726dec_ti. IXX
_VOL_IVOL = _VOL_TI_IVOL;	_G726DNC_IG726DNC = _G726DNC_TI_IG726DNC;

将 app.pjt 工程的预处理包括的搜索路径进行如下修改。

查 找	替 换	查 找	替 换
.. \algFIR	.. \algG726ENC	.. \algVOL	.. \algG726DNC

保存所有文件和工程，重新 build。在目标板上装载运行，以检查是否正确。

与 FIR 滤波器和 VOL 处理相比，应该听到有轻微衰减的音频。由于声音合成的压缩率，这个衰减是正常的。如果你一点声音都没听到，需要检查 thrAudioproc.c 中的 g726encParams 和 g726decParams 的设置。

2.2.3.7 RF3 的性能

RF3 的 CPU 负载性能如表 2.8 表示。表中第一行是使用了 FIR 和 VOL 算法的 CPU 负载值，后面两行的 CPU 负载值是通过简单地将源缓存直接拷贝到目标缓存的方法代替 FIR 和 VOL 算法处理的结果。可见，大部分的 CPU 时间花费在了处理算法上，RF3 只增加了很小的负荷。另外，设备驱动在算法消耗的 CPU 负载中占了很大的百分比。

表 2.8 RF3 的 CPU 占用

配 置	CPU 负载
TI 提供的 RF3 软件包	11.7%
减去 FIR 和 VOL 算法的两通道 RF3	4.7%
减去 FIR 和 VOL 算法的单通道 RF3	2.3%

CPU 的负载率是使用 STS 对象中的指令周期测量来计算的。在 CCS 中还可以用 CPU 负载图这种更精确的方法。具体测量条件如下。

- 100MHz 速率的 C5402 DSK。
- 采样率：8KHz。
- 每帧的样点：80。
- 优化标志位：无。
- 调试标志：-g。
- 所有关键代码和数据都在内部存储器中。

RF3 整个应用占用的存储器少于 16KB，因此可以用于 TMS320C5402。例如前面的例子中，RF3 框架占用 11.4KB，加上算法和数据一共 15.3KB。

表 2.9 所示为 C5510 DSK 上的 RF3 例子的存储器占用，没有进行优化。所有值的单位是 16 比特字。表 2.10 所示为其中基本模块的存储器占用，可以知道优化时如何裁剪。

表 2.9 RF3 例子的存储器占用

类 别	16 位字大小 (16 进制和十进制)	运行时代码大小 (RF 模块)
. DARAM\$heap	0x00 (3 072)	
. SARAM\$heap	0x2000 (8 192)	
. sysstack	0x400 (1 024)	
. stack	0x400 (1 024)	
rts	0x1a4 (420)	
ALGRF	0x161 (353)	0x49 (73)
hwiVec	0x80 (128)	
FIR	0x133 (307)	0xdf (233)
VOL	0xe7 (231)	0x98 (152)
Application	0x60e (1 550)	
PIO	0x26b (619)	0x146 (326)
driver	0x56a (1 386)	0x2ea (714)
CSL	0xc14 (3 092)	
UTL	0x26e (622)	0x18d (397)
LOG	0x16d (365)	
STS	0xc1b (203)	
RTDX	0x806 (2 054)	
DSP/BIOS kernel	0x1c8b (7 303)	
Total	0x7ccd (31 949)	

表 2.10 RF3 基本模块的存储器占用

总的大小	31 949 16 位字
未使用的内部堆	减去 2 680
未使用的外部堆	减去 8 084
应用程序缓冲区	减去 640
算法的堆	减去 365
VOL 算法	减去 231
FIR 算法	减去 307
FIR 系数表	减去 64
未用的栈和系统系统的栈	减去 1 712
调试对象 LOG	减去 365
调试对象 STS	减去 203
UTL 调试模块	减去 622
RTDX 模块	减去 2 054
所有的网络服务请求记录	减去 2 528
产生的基本框架的大小	12 094 16 位字

2.2.4 RF5——扩展型编程框架

RF5 的目的是方便设计者构建包含大量算法和通道的大规模应用程序。RF5 使用了具有阻塞态的线程 (tasks)，可用于包含线程间有复杂依赖关系的应用程序中。因此，RF5 更适用于含有多通道和复杂算法结构的高密集度系统应用。

RF5 使用“单元 (cell)”来描述对一个算法的封装。一个通道可以包含多个单元，也就是多个算法。RF5 提供了创建和控制单元和通道的模块，以及支持单元间的通信、同步和 XDAIS 共享内存的其他模块。RF5 单元中所使用的算法是符合 eXpressDSP 算法标准的，容易集成。

#### 2.2.4.1 RF5 概述

适合于使用 RF5 的典型应用包括大量的通道和算法需求，以及必需的控制能力。内存消耗对于 DSP 应用来说总是一个关键问题，基于 RF5 的应用会比低端 DSP 应用需要更多的存储空间。

RF5 包括基础软件，如 DSP/BIOS，片上支持库 (CSL)，以及 XDAIS。也调用一些参考框架模块提供的服务，例如，UTL 模块用于调试和诊断。开发者可以以现有框架为基础，或以相对简单的方式将自己的应用与更高级别的框架接口。

表 2.11 详细给出了 RF5 的特点，可以作为判断 RF5 是否适合目标应用的指南。

表 2.11 RF5 基本模块的存储器占用

设计参数	RF5	注 释
静态配置	√	
动态对象创建	√	
静态存储管理	√	
动态存储分配	√	
通道/信道	1 ~ 100 +	若 1 ~ 10 个通道，推荐采用 RF3
eXpressDSP 算法	1 ~ 100 +	若 1 ~ 10 个算法，推荐采用 RF3
最小存储器占用	×	
DSP/BIOS 实时分析	√	
DSP/BIOS 内核调度	√	主要使用 TSK 和 HWI 线程
片上支持库	√	
XDAIS 算法标准	√	
支持多种运行速率和优先级	√	
线程阻塞	√	
实现控制函数	√	
实现 DSP - GPP 函数	×	这将是 RF6 的关键特性

RF5 有以下一些关键特征。

- 可扩展的通道管理器。通道管理器使应用可以包括大量的 XDAIS 算法，同时减少系统中 TSK 的数量。算法间能高效地共享动态数据存储器，并且算法能非常容易地通过 ICELL 接口进行替换。
- 基于 TSK 的应用。RF3 是基于 SWI 的。与 RF3 不同，RF5 是基于 TSK 的。TSK 模块提供了比 SWI 模块更灵活的调度机制，但会消耗更多的时间和存储器。
- 高效的任务间通信。SCOM 模块可提供多任务间的简单、单向零拷贝数据传递。将 SCOM 作为一个标记传递器，获得标记的任务可以自由访问缓冲区。
- 结构化的线程安全控制机制。控制线程与外部的接口，例如通过读取板面上的设置改变应用流程，并且将控制信息传递到其他线程。
- 方便 I/O 设备驱动的替换。通过使用 RF5，IOM 模块可以将 mini 驱动连接到 DIO 适配器。
- 易于调试。RF5 定义良好的结构允许快速调试。而且，UTL 模块允许快速改变调试的级别。

RF5 可用于多种应用系统，图 2.32 所示为 RF5 应用的一个例子。其他典型应用包括无线基站、视频基础设备、交互式 TV 服务、通用端口转换等。

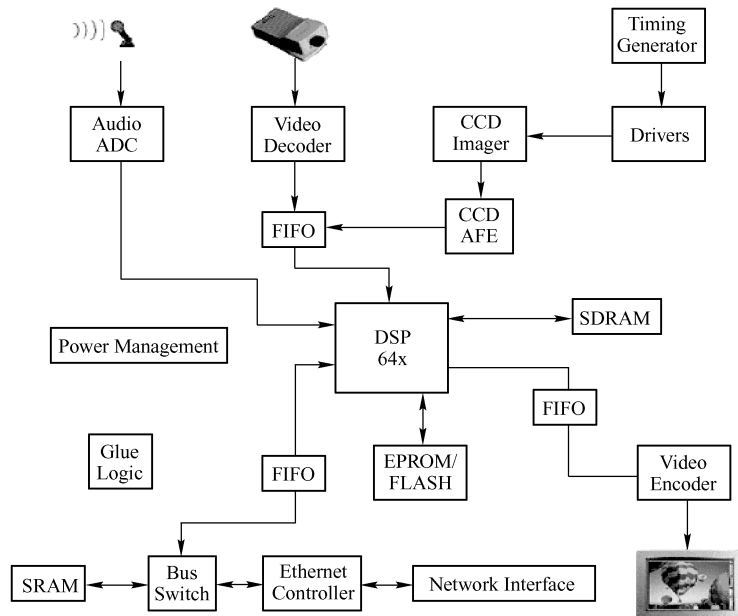


图 2.32 视频监控系统

#### 2.2.4.2 安装和运行 RF5

下面的步骤说明怎样将主机与硬件连接，以音频应用为例。

- 关闭你的 PC。
- 将正确的数据传输线连接到 DSP 板上。
- 将数据传输线的另一端连接到 PC 上正确的接口。
- 将音频输入设备比如麦克风或者 CD 的输出连接到板子上的音频输入。也可以将 PC 声卡的音频输出直接连接到板子上的音频输入。
- 将一个（或多个）扬声器或其他音频输出设备连接到板子的音频输出端。
- 给板子接通电源。
- 打开 PC。

下面列出运行 RF5 所需要的软件安装步骤。

- 安装 CCS2.2 以上的版本，推荐安装最新的版本，新版本会包括对已有问题的修正。
- 检查并口的配置，确保并口工作在 ECP 或 EPP 模式下，一般为 0x378 端口。检查并口配置的资料，参见随板子提供的快速指南。
- 使用 Setup CCS 程序配置板子需要的软件，具体细节参见板子提供的文档。
- 从 TI 网站（[www.dspvillage.com](http://www.dspvillage.com)）下载 RF 文件，将此文件解压缩。推荐放置于 CCS 安装目录下的 myproject 文件夹。不要将解压的文件放到 c:\Program Files 文件夹。
- RF 解压后的顶层文件夹叫作“referenceframeworks”，在以后章节中这个文件夹的路径用“RF\_DIR”代替。

创建和运行 RF5 应用程序的步骤如下。



- 确认 CCS 选择了与板子匹配的 GEL 文件。
- 在 CCS 中, 选择 Project→Open 打开 RF\_DIR\apps\rf5\projects\target 路径下的工程文件 app. pj1, 选择与 DSP 板子匹配的 target ( 比如, RF\_DIR \ apps \ rf5 \ projects \ teb6416 )。
- 选择 Project→Build 创建 RF5 可执行程序 app. out。  
注意: 如果使用新版本的 CCS, 最好在 MS - DOS 命令窗口运行 RF\_DIR\build. bat 这个批处理文件, 重建所有的 RF 工程, 以确保所有的模块与最新的 TI 代码生成工具同步。
- 选择 File→Load Program 将 Debug 文件夹中的 app. out 文件装载到目标板上。
- 启动 CD 或是其他的音频输入设备。
- 选择 Debug→Run ( 或者按 F5 ), 就会听见从连接在目标板上的喇叭中传出的 FIR 滤波后的音频输出。
- 选择 File→Load GEL, 从工程文件夹中选中 appControl. gel 文件。appControl. gel 文件是控制算法参数的 GEL 脚本。当改动这些控制时, 脚本向目标板的程序写入相应的参数。
- 选择 GEL→Process Control→Volume。会出现一个范围 0 ~ 200 的滑动条 ( 默认值是 200 ), 用于控制输出立体声解码器的输出音量。
- 选择 GEL→Process Control→Filter1 ( 或 Filter2 )。会出现一个值为 0 ~ 2 的滑动条 ( 默认值是 1 ), 用于控制 FIR 滤波器的系数。

### 2.2.4.3 RF5 文件结构

RF 的目录树包括应用源文件和库模块。图 2.33 显示了 RF5 的文件夹结构, 其中专门标出了一些重要的文件。主要的文件夹包括以下几类。

- apps\rf5 包含了组成 RF5 的 CCS 工程。如果要修改 RF5, 可以在同级文件夹下复制整个 rf5\ 目录树, 然后在这个复本上修改。包括的子文件夹有以下几个。
  - appConfig, 包含 DSP/BIOS 的配置脚本文件 appcfg. tcf。
  - cells, 包含应用程序的算法单元的实现代码, 系统集成者实现简单的“胶水代码”, 将 XDAIS 算法接口与 RF 的 API 匹配。例如, 你可以创建一个文件夹: cells\mp3, 用于存放 cellMp3. h, cellMp3encode. c 和 cellMp3decode. c。
  - projects, 包含硬件相关的 RF5 应用程序文件。其中有配置文件、工程文件、GEL 和连接器文件。这些文件放置在硬件平台名字的文件夹中。
  - threads, 包含与硬件独立的线程或任务源文件。
- include 包含在 RF 使用的一些公共头文件。RF5 使用其中一些, 但不是全部。算法和框架代码都要引用公共头文件。相反, 私有头文件和源代码一起储存, 这些源代码包含它们并且不会被其他模块使用。每个库模块在这个文件夹中都有一个头文件。
- lib 包含一些和 RF 应用连接的库文件。RF5 使用其中一些, 但不是全部。每个库模块在这个文件夹中都有和 DSP 芯片对应的一个库。
- src 包含模块的源文件。在每个文件夹中的 readme. txt 都提供了模块和它们用法的信息。库模块一般不需要修改。

RF5 的模块组织如图 2.34 所示, 包括以下几类:

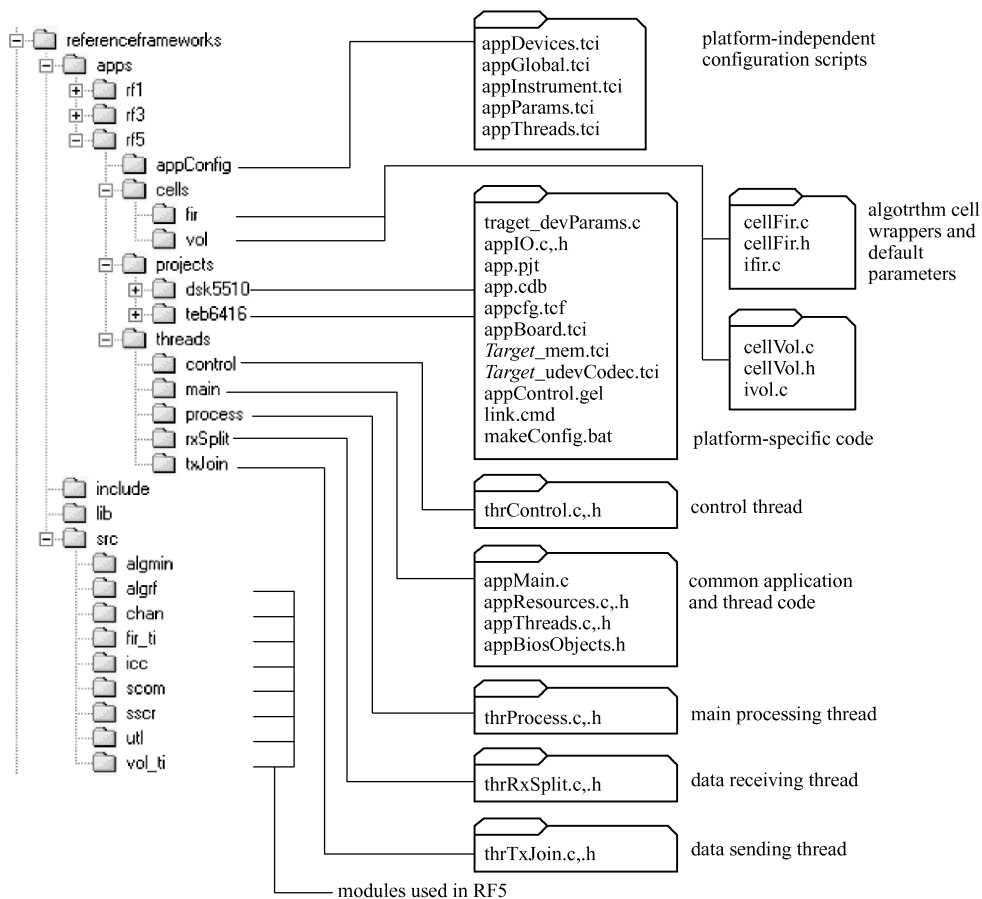


图 2.33 RF5 的文件夹结构

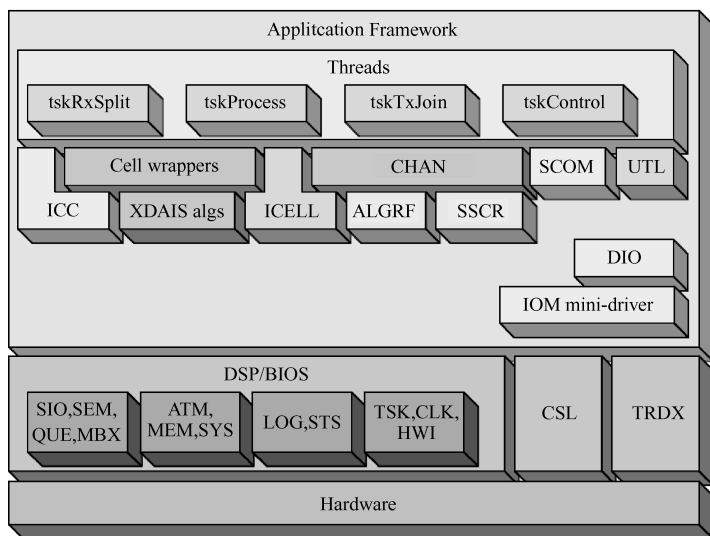


图 2.34 RF5 的模块组织

- 线程相关的处理，图中最上面一层的四个模块；
- Cell 的外壳，包括应用程序中 cell 的实现代码，也包括简单的“胶水代码”；

- XDAIS 算法, 即符合 eXpressDSP 算法标准要求的算法;
- ALGRF, 与 CCS 的 ALG 模块类似;
- CHAN, 管理 cell 中串行执行的 XDAIS 算法;
- SCOM, 实现任务间的同步消息传递;
- ICELL, 定义包装在 cell 中的算法的接口;
- ICC, 提供 cell 间的通信机制, 以便数据从 cell 输入和输出;
- SSCR, 管理 XDAIS 算法申请的片上动态存储器;
- UTL, 支持调试和诊断;
- DIO, 流设备适配器, 位于设备驱动的上层, 为 IOM mini 驱动和 SIO 数据流对象提供接口, 通过执行 DSP/BIOS cell 来管理缓冲区传输;
- IOM mini 驱动, 在线程和硬件设备之间提供低层设备驱动接口;
- DSP/BIOS, 是一个可扩充的实时内核, 包括线程调度、存储管理、监测和 I/O 等模块;
- CSL, 片级支持库。

#### 2.2.4.4 RF5 应用系统设计方法

首先简要介绍 RF5 如何进行数据处理、数据通信以及传递控制消息的。

RF5 的四个基本的数据处理组件为任务 (task)、通道 (channel)、单元 (cell) 和 XDAIS 算法。顶层是 DSP/BIOS 任务, 一个任务可包含多个通道, 一个通道可包含多个单元, 一个单元封装一个 XDAIS 算法。每个组件有多个实例。例如, 可以每一个通道使用滤波器算法的一个实例, 分别使用不同的滤波参数以及不同的历史记忆。每个实例的数据描述是不同的, 但对数据操作的代码是相同的。通道和单元也是如此。通常当我们提到一个组件, 事实上是指它的一个实例。

XDAIS 算法有标准的资源管理函数 (申请存储器和 DMA)。然而, 实际的数据处理功能是算法的核心, 却没有标准的命名规则。单元的目标是通过定义一个处理函数, 在算法和外部之间提供一个标准界面。每个单元实现一个简单的 ICELL 界面, 共定义了四个功能: 打开、执行、关闭和控制。除执行外, 其他功能是可选的。

通道包括若干单元, 目的是连续执行它的单元。通道不需要写额外的代码。典型的几种通道包含一批单元实例, 通过不同的参数来实现特定的功能。

任务可包括若干通道, 并且连续地执行它们。任务的目标是通过设备驱动器以及其他任务在上层组织数据通信。与通道不同, 任务为用户自己写的专门代码。这些代码通常只用来接收或输出数据, 并且控制通道执行。一个任务也可能没有任何通道。

RF5 的数据通信组件分为任务级数据通信和单元级数据通信。任务级数据通信使用 SIO 对象和 SCOM 消息。对于单元级数据通信, 采用 ICC 对象。ICC 对象的目的是描述一个单元读取数据的缓冲区, 或者是单元写入数据的缓冲区。每个单元有一份输入对象列表和一份输出对象列表。两个单元通过拥有同一个 ICC 对象列表就可以高效地进行通信。

RF5 的数据通路如图 2.35 所示, 数据通路从设备驱动开始, 输入 RxSplit 线程的 double 型缓冲区 bufRx 中。RxSplit 线程不直接与设备驱动通信, RxSplit 拥有一个 SIO 输入流对象, 通过 DIO 模块与 IOM mini-driver 接口。通过 SIO, RxSplit 在需要时告诉设备驱动它所需要数据的地方。因为是双缓存, mini-driver 能保证在 RxSplit 处理一半准备好的数据的同时,

从 bufRx 中抽取数据的下一个一半。

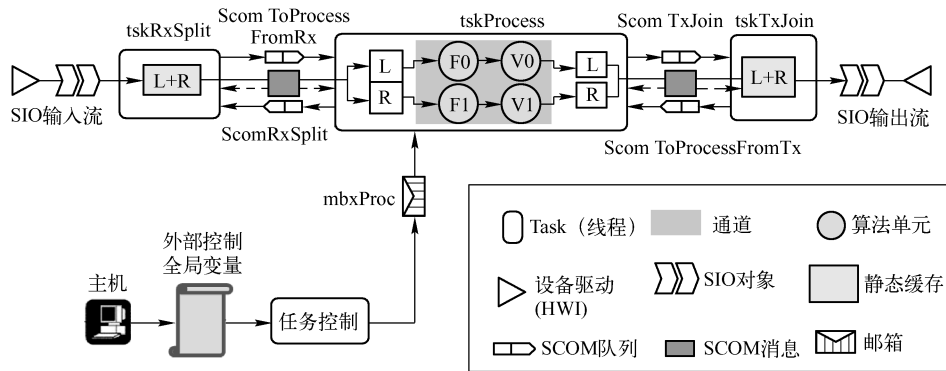


图 2.35 RF5 的数据通路

当通过 SIO 接收到输入帧，RxSplit 会从它的 scomRxSplit 队列收集 scomMsgRx SCOM 消息，如果没有消息，则阻塞等待。此消息描述了处理任务拥有的两个缓存。RxSplit 向缓存写入各自的通道数据。然后，RxSplit 将 scomMsgRx 消息放回到处理线程的接收 SCOM 队列。

当处理线程收到 scomMsgRx 消息，给处理线程发送一个信号，表示新的输入帧已经被分离成了通道并存入了处理线程的私有缓存 bufInput [NUMCHANNELS] 中。这两个缓存与 FIR 单元的输入 ICC 对象连接，以便线程执行它的 NUMCHANNELS (2) 通道。在每个通道中，FIR 单元从相关的 bufInput 缓存中读取，处理数据，并存入 bufIntermediate 缓存。VOL 单元从此缓存中读取数据，并将结果存到通道的 bufOutput 这部分缓存中。幸亏使用了 ICC，单元不需要知道实际的缓冲区名字，所以它们能在任何通道和任务中运行。

因为 RF5 的结构如此，所以非常容易去改变通道的数量。对每一个通道，你能指定它所包含的单元，以及它执行的 XDAIS 算法。

RF5 使用两种 DSP/BIOS 线程：硬件线程中断 (HWI) 和任务 (TSK)。DSP/BIOS 在初始阶段调用用户的 main () 函数。

如图 2.35 所示，RF5 任务在无限循环中等待数据，处理数据并且传递数据到其他线程。任务使用 SIO 对象（与设备驱动相关）、SCOM 对象（在任务间传递消息），以及 MBX 对象（传送和接收控制消息）等实现同步。

注意 RF5 应用代码不直接使用旗语，而是把旗语隐藏在 SCOM、SIO 和 MBX 对象中，因此使用起来更安全。推荐 RF5 应用程序尽可能少地用任务和旗语，因为基于旗语的同步是造成运行问题的主要原因。幸运的是，RF5 结构就是在一个任务中实现多通道、多单元和多种算法。

通常在多任务和多优先级系统中，以不同的速率处理数据非常重要。例如，一个电话系统可能有一个高优先级的任务处理 10ms G.729 通道，一个中等优先级任务处理 GSM 通道，一个低优先级的任务处理 30ms G.723 通道。RF5 中，默认情况下所有任务优先级是相等的。

下面分别介绍四个线程：RxSplit、TxJoin、Process 和 Control。

#### 1) RxSplit

这个线程将信号分到应用指定数量的通道中。线程的源代码在 thrRxSplit.h 和 thrRxSplit.c 文件中。

在初始化和启动阶段，这个线程创建并打开一个 SIO 输入对象 “inStream”，静态分配

缓冲区，创建用于接收的 SCOM 队列“scomRxSplit”。

在运行期间，tskRxSplit 任务调用 thrRxSplitRun() 函数。在函数进入无穷循环前，线程打开两个 SCOM 队列以获得它们的句柄。在无穷循环中，此函数使用 SIO\_reclaim() 函数从 inStream 请求一个满缓冲区。如果没有准备好的缓冲区，则阻塞（等待）。然后线程通过调用 SCOM\_getMsg()，等待一个来自 Process 线程的 SCOM 消息（同样地，如果消息不在队列中则阻塞）。SCOM 消息结构简单，在 appTheads.h 中定义的 ScomBufChannels 包含了指向每个目标缓存的指针。

一旦线程接收到 SCOM 消息，任务循环通过通道和帧单元将输入数据分解成 SCOM 消息指定的多个缓冲区。接着调用 SCOM\_putMsg() 函数将信息放入 Process 线程的接收队列“scomToProcessFromRx”中，意味着分离的数据已经可用。最终，调用 SIO\_issue() 将一个空缓冲区送回 inStream。

RF5 中使用这些任务来实现预处理和后处理的快速模型。它们允许 RF5 使用一个通用的设备驱动。这是个重要的优点，因为系统集成者可能不是一个有经验的设备驱动开发者，只要不修改驱动，系统就可以快速建立。

## 2) TxJoin

这个线程与 tskRxSplit 很相似，但它是把分开的信号合起来。其源代码在 thrTxJoin.c 和 thrTxJoin.h 文件中。

在启动阶段，RxSplit 和 TxJoin 都通过它们各自的 SIO 对象将缓冲区分发到设备。重要的区别是：RxSplit 的 SIO 对象是输入对象，所以 RxSplit 给各设备分发空的缓存，并且等到驱动将它们填满数据后收回它们。而 TxJoin 的 SIO 对象是输出对象，所以当它通过 SIO 分发给设备两个缓冲区时，设备立刻启动输出（静默时以 0 填满缓冲区）。TxJoin 必须预先填充传输端缓冲区，以确保连续的输出和避免开始会听到砰砰声和嘀嗒声。

## 3) Process

这个线程是应用程序的核心，它通过 XDAIS 算法处理数据。若希望 RF5 应用能满足你的需求，这是需要修改的主线程。其源代码在 thrProcess.c 和 thrProcess.h 文件中。

thrProcess.h 中定义的 ThrProcess 结构如下。

```
typedef struct ThrProcess {
    CHAN_Obj      chanList[ NUMCHANNELS ];           //array of channel objects
    ICELL_Obj     cellList[ NUMCHANNELS * NUMCELLS ]; //array of cell objects
    Sample        * bufInput[ NUMCHANNELS ];        //pointers to input buffers
    Sample        * bufOutput[ NUMCHANNELS ];        //pointers to output buffers
    Sample        * bufIntermediate;                //pointers to intermediate buffers
    ScomBufChannels scomMsgRx;                       //SCOM object
    ScomBufChannels scomMsgTx;                       //SCOM object
} ThrProcess;
```

此结构的前面两部分是通道和单元的实际占位符。数据缓冲区在结构外静态分配空间，因为通常它们可能需要按缓存目标排列对齐，而结构成员很难控制对齐。结构里只存放了指向缓冲区的指针。

技术上，不需要清楚任务的状态，因为每个任务都有自己的栈。然而，有一个线程相关



的变量在调试中全局可见，很有用，并且避免静态变量是一个好的编程习惯（尤其是考虑到可重入过程）。

线程在 `thrProcessInit()` 函数中初始化单元矩阵，并且在 `thrProcessStartup()` 中创建实际的 XDAIS 算法和初始化通道队列。需要两个阶段的初始化，因为在第一阶段，系统收集关于所有单元的信息，计算最小动态存储的大小，而没有创建任何东西。在第二阶段，分配存储器并且创建算法。

大多数的启动工作在 `setParamsAndStartChannels()` 函数中实现。通过一个参数，这个函数知道是 `thrProcessInit()` 还是 `thrProcessStartup()` 调用了它。也就是说此函数知道它是在第一阶段还是在第二阶段被调用。在第二阶段，它定义了单元对象的内容并且注册了它。在第二阶段，它通过创建 XDAIS 算法初始化通道对象。但在两个阶段中，它都需要为单元中的算法设置 XDAIS 参数。默认的 XDAIS 参数是由全局对象来定义的，但通常需要修改一些参数值。为了避免永久地修改这些只在初始和启动阶段需要修改的参数，这个函数通过将全局参数对象拷贝到局部参数对象来修改。接着改变需要的值，调用单元注册器或通道创建函数（根据阶段）。局部对象在处理完成后被释放。

除了定义单元对象和算法参数外，初始阶段也创建 SCOM 对象，并且初始单元的内部传递缓存（ICC）通过 SCOM 对象指向静态数据缓冲区。

Process 线程将描述输入缓存 `scomMsgRx` 的消息放在 `RxSplit` 的接收队列中。而把描述输出缓存 `scomMsgTx` 的消息放在自己的接收队列中。这使得初始循环简单明了：得到两个 SCOM 信息，处理数据，将两个 SCOM 信息放入其他线程队列中，依次重复。

主循环首先检查 `mbxProcess` 邮箱是否包含了控制信息。如果邮箱为空，它将不等待。接着，它两次调用 `SCOM_getMsg()` 函数来从 `scomToProcessFromRx` SCOM 队列中得到一个满输入缓冲区，以及一个从 `scomToProcessFromTx` SCOM 队列的空输出缓冲区。若消息没有准备好，它在等待时将阻塞。

```
//Main loop
while ( TRUE ) {
    scomBufChannels * scomMsgRx, * scomMsgTx;
    //check for control (MBX) messages (not to be confused with scom msgs)
    checkMsg();
    //get the message describing full input buffers from Rx
    scomMsgRx = SCOM_getMsg( scomReceiveFromRx, SYS_FOREVER );
    //get the message describing empty output buffers from Tx
    scomMsgTx = SCOM_getMsg( scomReceiveFromTx, SYS_FOREVER );
    //record the time period between two frames of data in stsTimeo
    UTL_stsPeriod( stsTimeo );
    //process the data
    for( chanNum = 0; chanNum < NUMCHANNELS; chanNum ++ ) {
        CHAN_Handle chanHandle = &thrProcess. chanList [ chanNum ];
        //Set the input ICC buffer for FIR cell for each channel
        ICC_setBuf( chanHandle -> cellSet[ CELLFIR ]. inputICC[ 0 ] ),
            scomMsgRx -> bufChannel[ chanNum ],
            FRAMELEN * sizeof( sample ) );
    }
```



```

//Set the output ICC buffer for VOL cell for each channel
ICC_setBuf( chanHandle -> cellSet[ CELLVOL]. outputICC[0] ),
           scomMsgTx -> bufChannel[ chanNum ],
           FRAMELEN * sizeof( Sample ) );

//execute the channel
UTL_stsstart( stsTime1 );           //start the stopwatch
rc = CHAN_execute( chanHandle, NULL );
UTL_assert( rc == TRUE );
UTL_stsStop( stsTime1 );           //elapsed time goes to this STS
}

//send the message describing full output buffers to Tx
SCOM_putMsg( scomSendToTx, scomMsgTx );
//send the message describing consumed input buffers to Rx
SCOM_putMsg( scomSendToRx, scomMsgRx );
}

```

#### 4) Control

每个线程都能通过邮箱来发送和接收控制信息。在 RF5 中，只有 Process 线程接收消息，而控制线程传送消息。大多数应用都有类似控制线程这样的线程。其源代码在 thrControl. h 和 thrControl. c 文件中。

控制线程与外部交互，例如，读取在应用运行时可改变的硬件设置值，并向其他线程（Process 线程）传递控制消息。在 RF5 中，控制信息流如图 2.36 所示。控制线程读一个全局变量，用户通过 GEL 脚本修改此变量，变量中包含通道数的信息以及使用的滤波器类型。每 100 个时钟周期，控制线程检查这个全局变量，如果它的值改变了，就向 Process 线程发送一个消息。appThreads. h 中定义了控制消息的格式，由 3 个 32 比特的无符号整数组成：消息命令、参数一和参数二。消息接收者 Process 线程在头文件中定义了它所接收的消息种类，控制线程负责将外部事件描述为合适的消息并传递。

前面提到过 RF5 任务使用 SCOM 模块进行任务间消息传递。SCOM 是一个通用任务间消息传递机制。SCOM 消息是用户定义的用于任务间通信的数据结构。一个任务通过 SCOM\_putMsg() 函数将 SCOM 消息放到 SCOM 队列上，或通过 SCOM\_getMsg() 函数从 SCOM 队列中取出消息。

一个 SCOM 队列是一个命名的、基于旗语的队列（使用 DSP/BIOS QUE 模块）。通过 SCOM\_create() 创建 SCOM 队列，队列的名字是一个字符串。任何知道一个 SCOM 队列名字的任务通过调用 SCOM\_open() 函数传递已知名字，就可以得到此队列的句柄（写入或读取数据需要此句柄）。典型地，每个作业创建一个或多个接收消息的 SCOM 队列。

RF5 以一种简单的方式使用 SCOM：Process 任务有一些缓冲区，由 RxSplit 任务写入数据，并被 TxJoin 任务读。Process 任务只需要：1) 告知其他任务缓冲区的地址；2) 确保两个任务不在同一时间进入同一缓冲区。所以 Process 任务分别为 RxSplit 和 TxJoin 各创建一个 SCOM 消息用于同步。Rx 消息描述了 RxSplit 应写入的缓冲区；而 Tx 消息描述了 TxJoin 应读取的缓冲区。每个 SCOM 消息变成了对应缓冲区的标记：拥有标记的作业能自由访问缓存。

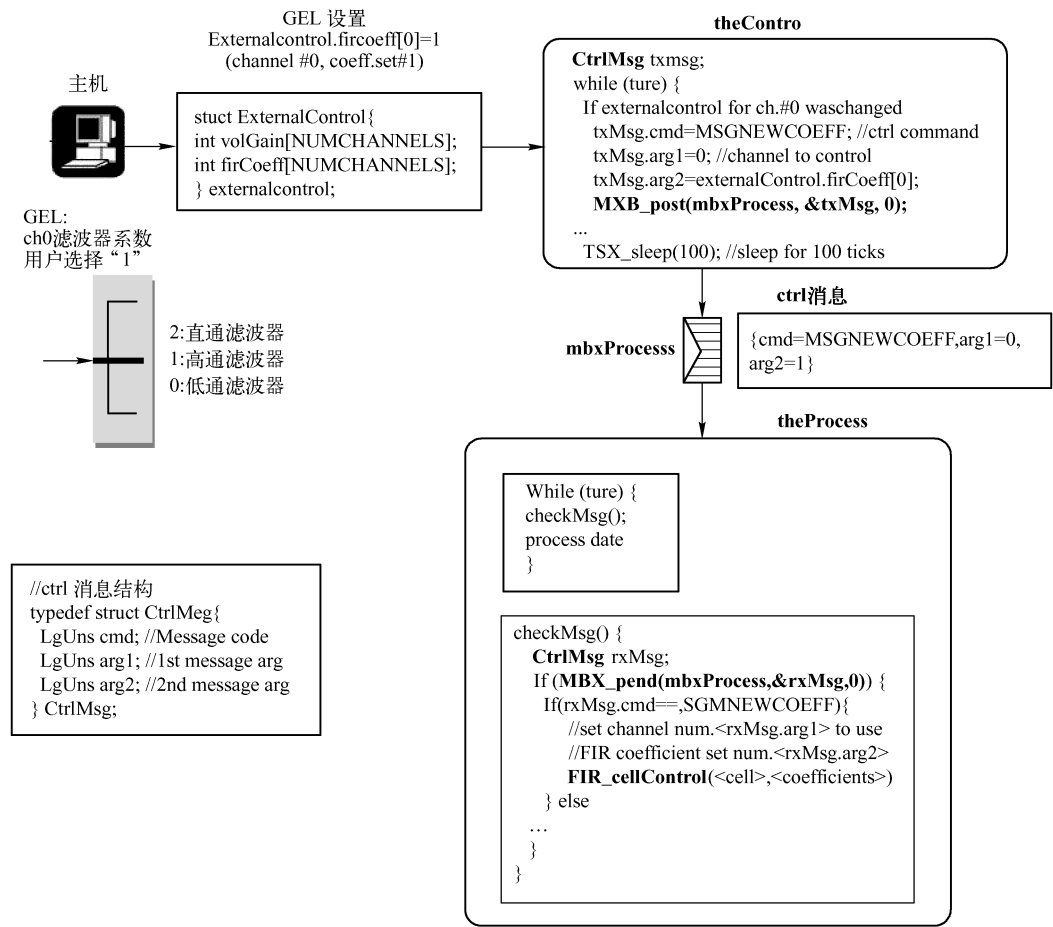


图 2.36 RF5 的控制信息流

2.2.4.5 RF5 的性能

RF5 的 CPU 负载性能如表 2.12 所示。可见，大部分的 CPU 时间花费在了处理算法上，RF5 只增加了很小的负荷。另外，设备驱动在算法消耗的 CPU 负载中占了很大的百分比。

表 2.12 RF5 的 CPU 负载

配 置	CPU 负载
TI 提供的 RF 5 软件包	6%
减去 FIR 和 VOL 算法处理的 RF5	5%

虽然 RF5 不以最小的存储器占用为目的，但存储器占用总是很受关注。RF5 的存储器占用（程序和数据）以及能给算法的空间见表 2.13。在此表中，没有计算应用相关的组件，包括算法、缓存和算法使用的堆，以及没有使用的栈和堆空间。RF5 基本模块的存储占用见表 2.14。

表 2.13 RF5 的存储器占用

	C5510 DSK	C6416 TEB
TI 提供的 RF 5 软件包	45 848 16 位字	59 011 16 位字
减去算法和具体应用组件的 RF 5	17 328 16 位字	27 993 16 位字

表 2.14 RF5 基本模块的存储器占用

总的大小	45 484 16 位字
未使用的内部堆	减去 3 300
未使用的外部堆	减去 8 088
应用程序缓冲区	减去 1 040
算法的堆	减去 365
VOL 算法	减去 202
FIR 算法	减去 278
FIR 系数表	减去 96
Cell 算法代码	减去 200
未用的栈和系统的栈	减去 2 622
未用任务的栈	减去 5 901
调试对象 LOG	减去 621
调试对象 STS	减去 163
UTL 调试模块	减去 630
RTDX 模块	减去 2 054
所有的网络服务请求记录	减去 2 960
产生的基本框架的大小	17 328 16 位字

## 2.3 RF 应用举例——网络数字监控系统

本节以基于 TMS320DM642 的网络数字监控系统为例，介绍 RF 在实际系统中的应用。

### 2.3.1 系统框图

系统采用 DSP 处理芯片 TMS320DM642（以下简称 DM642），配以 Decoder 视频解码、Ethernet 接口、MCU、GPIO 和 RS232/RS485 异步串口等多种系统外设，形成智能型网络数字监控平台，其结构框图如图 2.37 所示。

系统的输入部分由模拟摄像机、视频放大模块、模拟视频显示模块和视频解码模块组成。系统可支持两路模拟视频输入，任意一路模拟视频输入分成两路视频信号。一路经视频解码模块转化成数字视频信号后，送入 DM642 的 VP 接口，然后存储在扩展外部存储器 SDRAM 中；另一路经过一级视频放大器后送入到模拟输出终端，用以观察原始的模拟视频输入信号，并以此来确定被监视现场图像是否实时地由前端模拟摄像机送入。

当数字视频信号存储在 SDRAM 后，运行 DM642 内的算法库，便可以对视频输入信号进

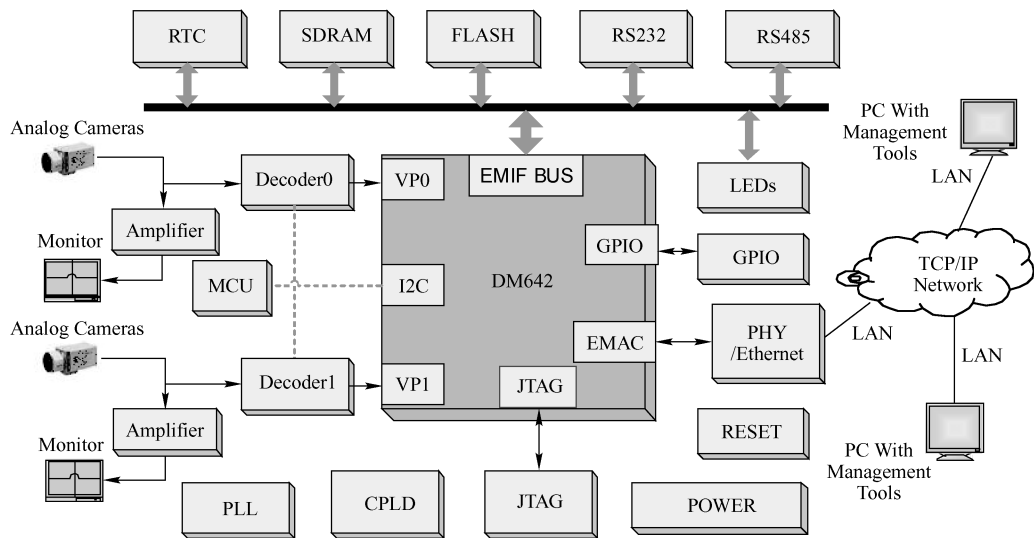


图 2.37 网络数字监控系统框图

行分析和处理，以及压缩编码。最终，经过压缩编码后的视频和其他信号由运行在 DM642 内的 TCP/IP 协议栈进行打包，再经 EMAC 模块传送到互联网 Internet 上，供远程的用户接收。实时运行在 DM642 内的算法库通过分析和处理视频输入信号来确定被监控现场是否有异常，一旦发生异常，则可以通过异步串口 RS232/RS485 模块或者 GPIO 模块触发相应的警报信息。

远程的终端用户也可以通过运行在 PC 端的 Management Tools 设置相应的参数，然后将这些参数送到 DM642 内进行相应的配置。比如说，可以设置图像编码的压缩比、相应的规则供算法库参考。

### 2.3.2 系统软件设计

基于 RF5 软件参考框架，系统软件的整体设计如图 2.38 所示。

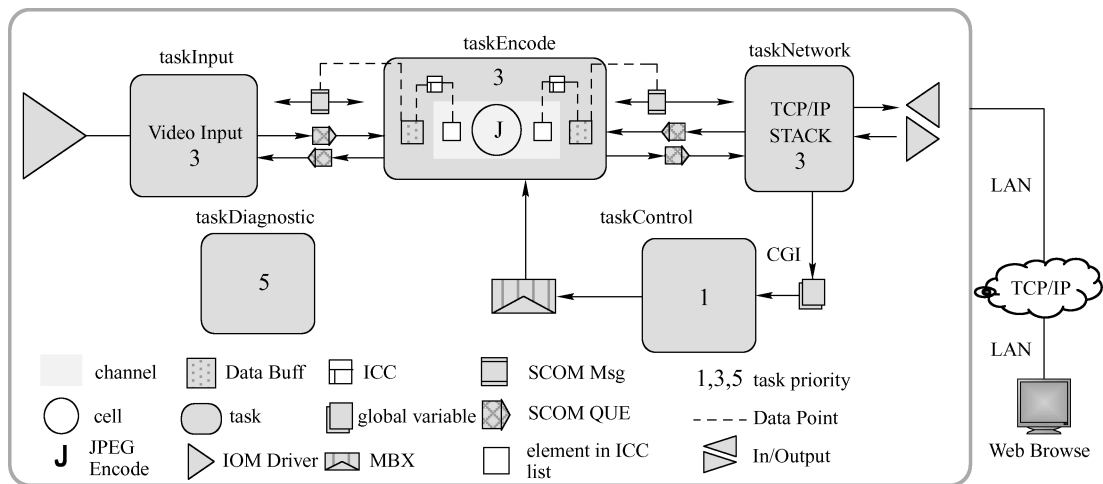


图 2.38 网络数字监控系统软件总体结构

系统可以支持两路摄像头的视频图像输入,然后分别对它们进行算法处理,再通过网络模块将处理过的视频图像发送到客户端,供远端的用户进行监控。如果算法检测出被监控现场有异常,可以网络进行报警,也可以通过板上其他外设进行报警。

DSP 端软件基于 DSP/BIOS 构架进行多线程调度和管理,在此基础上,使用基于 RF5 的同步通信模块 SCOM 通信,以便开发高密度的多通道、多算法系统应用软件。智能监控算法的集成遵循 XDAIS 算法标准,增强了智能监控算法的可移植性,减少开发周期和成本。

根据功能的不同,该软件模块分为 5 个部分:图像处理、参数控制、模拟摄像头视频输入、基于 PCI 总线的接收和发送数据。

数据通过 PCI 总线从设备管理模块发送到本模块后,PCI 接收线程根据 WBinfo 中的数据信息,将数据搬到指定的存储空间,如果是图像数据,通过 SCOM1 将数据的信息(如起始地址、长度、类型等)发送给图像处理线程。

图像处理线程集成并运行遵循 TI 推荐的 TMS320 DSP 算法标准的智能监控算法。该线程首先通过 SCOM1 (或 SCOM3) 获取图像数据,并动态申请内存空间存储结果,通过 SCOM2 通知 PCI 发送线程需要发送的数据。但其并不等待 PCI 发送线程的应答,直接进行下一次算法运算。这样可以有效地避免 PCI 半双工工作模式导致 DSP 端线程阻塞的情况。

PCI 发送线程检测到 SCOM2 不为空后,将数据信息填到 RBinfo 中,然后向设备管理模块发送传输请求中断。通过 PCI 总线完成数据传输后,该线程释放内存,再次检查 SCOM2。

控制线程检测 PCI 接收线程接收到的数据类型,如果是参数、控制命令等,该线程以邮箱 (MailBox) 的形式通知图像处理线程。

模拟摄像头视频输入是本模块可选功能。当指定模拟摄像头作为 DM642 PCI 卡的视频信号源时,模拟摄像头视频输入线程调用 IOM 驱动提供的 API 函数获取数字图像帧,存入指定的存储空间,并通过 SCOM3 发送给图像处理线程。等待图像处理线程返回 SCOM3 后,接收下一帧图像。

### 2.3.3 算法集成到 RF5

基于 RF5 参考框架的系统软件设计只需要针对具体的系统应用做以下修改:(1)设计整个系统的线程(task)和通道(channel);(2)更换底层硬件的驱动程序(IOM - Driver);(3)更换具体的数据处理算法(XDAIS)。

如图 2.38 所示,本软件一共创建了 5 个线程(tasks)和 1 个通道(channel)。其中 taskDiagnostic 线程的优先级最高为 5,taskInput、taskEncode 和 taskNetwork 的优先级为 3,taskControl 的优先级最低为 1。taskDiagnostic 线程主要完成系统各硬件模块的检测,不会与系统其他线程进行通信。taskInput、taskEncode、taskNetwork 和 taskControl 线程是系统的核心线程,不断地完成从底层视频驱动获取视频信号,将视频信号进行 JPEG 编码,再通过网络传给远程用户进行显示。taskInput、taskEncode 和 taskNetwork 线程之间通过同步通信模块(SCOM)进行同步和通信,taskControl 线程和 taskEncode 线程之间通过邮箱(MBX)进行通信。下面详细介绍各个线程的功能。

#### 1) taskProcess 线程

图像处理的主线程。

#### 2) taskInput 线程

系统完成视频驱动程序初始化以后,主要调用 taskInput 线程来从视频驱动程序中获得

一帧需要处理的视频图像，同时将 taskEncode 线程处理完的视频图像 buff 返回给底层驱动以便采集新的一帧视频图像。由于在整个系统软件中只定义了一个通道（channel），因此为了完成同时对两路 D1 格式视频输入的 JPEG 编码，将第二路视频输入信号进行重采样，转化成 CIF 格式，并且通过 QDMA 搬移叠加在第一路 D1 格式视频输入的右上方，形成“画中画”效果。

### 3) taskEncode 线程

系统主要调用 taskEncode 线程来完成对视频输入图像的 JPEG 压缩编码，以便在网络上对视频图像流进行传输。在这个线程中目前只初始化了一个通道（channel），并且在这个通道中包含了一个封装了 TI 公司 XDAIS JPEG 算法库的内核（cell）。

接下来介绍如何将 JPEG 算法封装到内核（cell）中，以及最后注册到通道（channel）里面。一个符合 TI 公司推荐算法标准（XDAIS）的算法提供了一个标准的接口，在具体的应用程序中需要将这个接口填入到内核的内部接口（ICELL）对象，接下来算法的打开、执行、控制和关闭都由内核接口（ICELL）对象来操作。

内核接口对象的数据结构如下。

```
typedef struct ICELL_Obj {
    Int          size;           /* 结构的大小 */
    String       name;          /* 被封装算法的名字 */
    ICELL_Fxns   * cellFxns;     /* 指向具体的 Cell 执行函数 */
    IALG_Fxns    * algFxns;      /* 指向算法的执行函数 */
    IALG_Params  * algParams;    /* 指向算法的参数 */
    IALG_Handle  algHandle;      /* 指向算法的句柄 */
    Uns          scrBucketIndex; /* 为算法分配的缓存索引 */
    ICC_Handle   * inputIcc;     /* 输入 ICC 对象列表 */
    Uns          inputIccCnt;    /* 输入 ICC 对象个数 */
    ICC_Handle   * outputIcc;    /* 输出 ICC 对象列表 */
    Uns          outputIccCnt;   /* 输出 ICC 对象个数 */
} ICELL_Obj;
```

将 JPEG 算法填入到 ICELL 对象的代码如下。

```
ICELL_Obj    * cell;
cell->name    = "JPEGENC";
cell->cellFxns = &JPEGENC_CELLFXNS;
cell->algFxns  = (IALG_Fxns *) &JPEGENC_IJPEGENC;
cell->algParams = (IALG_Params *) &IJPEGENC_PARAMS;
cell->scrBucketIndex = THRIOSSCRBUCKET;
inputIcc     = (ICC_Handle) ICC_linearCreate( NULL, 0 );
outputIcc    = (ICC_Handle) ICC_linearCreate( NULL, 0 );
```

将封装了 JPEG 算法的内核注册到通道中。

```
CHAN_regCell( cell, &inputIcc, 1, &outputIcc, 1 );
```



通过指定输入/输出 ICC 对象所指向的数据 buff 和通道执行函数的执行, 便可以利用 JPEG 压缩算法对视频图像进行压缩了, 代码如下。

```
//将输入 ICC 对象指向存储原始视频图像的数据 buff
inBuf[0] = pMsgBuf -> bufY;
.....
ICC_setBuf( chanHandle -> cellSet[0]. inputIcc[0],
            inBuf, sizeof( void * ) * 3 );
//将输出 ICC 对象指向存储 JPEG 视频图像的数据 buff
outBuf[0] = &jpg_size;
outBuf[1] = jpg_img;
ICC_setBuf( chanHandle -> cellSet[0]. outputIcc[0],
            outBuf, sizeof( void * ) * 2 );
//调用 JPEG 算法对视频图像进行处理
CHAN_execute( chanHandle, frame_num );
```

taskEncode 线程也是通过 SCOM 通信机制与 taskInput 线程和 taskNetwork 线程进行同步和通信的。将编码后的 JPEG 图像数据指针封装在一个 scomMsg 中, 和一个 SCOM 消息一起发送给 taskNetwork 线程。同时, 将处理完的视频图像数据指针也封装在一个 scomMsg 中和一个 SCOM 消息一起发送给 taskInput 线程。taskEncode 线程通过邮箱 (MBX) 与 taskControl 线程进行通信, taskEncode 线程中每次对视频图像信号进行 JPEG 编码前, 都会对压缩比参数进行确认, 该压缩比参数是否改变。如果有改变, 就用新的参数来进行编码。而改变的参数则来自于 taskControl 线程。

#### 4) taskNetwork 线程

在网络调度线程 (taskNetwork) 中有两部分功能, 第一是初始化 NDK 中的 TCP/IP 协议栈, 并且根据系统的应用配置好协议栈的 DHCP 和 Http Server 等功能; 第二是创建一个新的发送线程, 并且根据创建的套接字 Socket, 不停地对网络进行监听, 一旦发现套接字上有请求就将动态生成的 Jpeg 文件发送给客户端。

在任何的网络程序被执行之前, 必须在网络调度线程 (taskNetwork) 中对 TCP/IP 协议栈进行初始化和做相关的配置。在 taskNetwork 线程中, 通过调用网络控制模块 (NETCTRL. LIB) 的函数, 完成对 TCP/IP 协议栈的初始化和配置。然后, 网络调度线程就可以通过调用网络控制模块 (NETCTRL. LIB) 的 NC\_NetStart 函数开启具体的网络应用线程了。注意, 任何具体的网络应用程序都不会在网络调度线程中被执行, 需要开辟新的线程来执行; 同时网络调度线程也只有所有的网络应用程序都被执行完以后才会结束。通过 NC\_NetStart 函数, 本系统动态地创建了一个 tskNetworkTx 网络数据发送线程。在 NC\_NetStart 函数中, 除了配置信息的句柄外剩下的三个参数均是指向具体函数的指针, NetworkOpen() 和 NetworkClose() 函数只有在网络系统初始化和结束的时候被调用一次。而 NetworkIPAddr() 函数则可以被多次调用, 每次当 IP 地址被添加到网络系统或者被移除网络系统, 都会调用此函数。因此非常适合网络系统 DHCP 的应用。

#### 5) taskControl 线程

在 taskNetwork 线程中, 系统配置了 http server 服务, 并且在 NDK 的嵌入式文件系统 (embedded file system EFS) 中创建和声明了一个 http 网页文件。在这个 http 网页文件中,

除了包含用以显示 JPEG 图像的 JavaApplet 小程序以外, 还编写了一个表单并配合一个 CGI 程序一起用来接收客户对 JPEG 压缩比参数的设置。当客户修改了 http 网页中的参数以后, 参数便会通过相应的 socket 传送给 CGI 程序进行处理和动作, 并将结果返回给客户端。这个 CGI 程序的作用就是将远程客户对 JPEG 压缩比参数的设置, 填入到整个系统程序的 JPEG 压缩比全局变量中。而 taskControl 线程每隔一定的时钟周期便会检查 JPEG 压缩比全局变量是否有更改, 如果发现有更改, 便通过邮箱机制 (MBX) 将客户修改的参数发送给 taskEncode 线程。

#### 6) 视频解码器驱动程序的集成

直接将视频解码器的驱动程序库添加到系统软件中, 然后设置好相应的参数, 并且利用类驱动层中的 VCAP 接口函数便可以获得驱动程序中的视频图像信号。

### 2.3.4 软件流程

整个系统软件的流程如图 2.39 所示, 系统代码会存储在 FLASH 当中。系统上电完成自启动 (Bootloader) 后, 实时操作系统 DSP/BIOS 会首先初始化, 随后在系统硬件初始化 (包括 EMIF 总线的配置和片上外设的设置等) 完成后进入主函数。

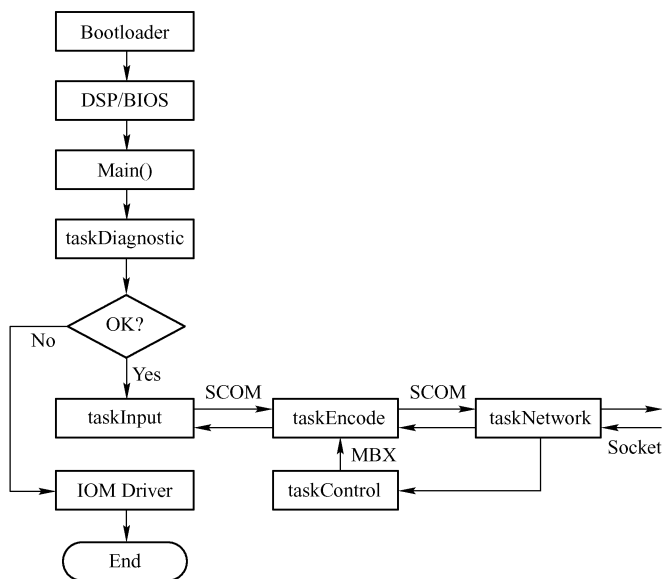


图 2.39 网络数字监控系统软件流程

在主函数 main() 中, 主要完成片上 Cache 的设置、RF5 和 XDAIS 算法标准, 以及各个线程 (task) 的初始化、SCOM 队列的创建等, 如图 2.40 所示。

在 main() 函数执行完以后, 实时操作系统 DSP/BIOS 便会接管系统的控制, 按照线程的优先级, taskDiagnostic 线程首先会获得系统资源而运行。在 taskDiagnostic 线程中, 先会对系统各种外设做一个检查, 只有自检通过以后其他线程才会获得系统资源而运行。否则, taskDiagnostic 线程便会通过调用类驱动函数释放掉系统所占用的资源, 退出整个程序的执行。

系统各种外设自检都通过以后, taskInput 线程便获得系统资源而运行。它首先从底层驱动获得一帧视频图像, 将图像进行简单的处理以后, 便将存储图像 buff 的指针随同一个 SCOM 消息发送到 taskEncode 线程的 SCOM 接收队列。随后, taskInput 线程将由运行态转为

挂起态，等待来自 taskEncode 线程的 SCOM 消息将其激活。具体的软件流程如图 2.41 所示。

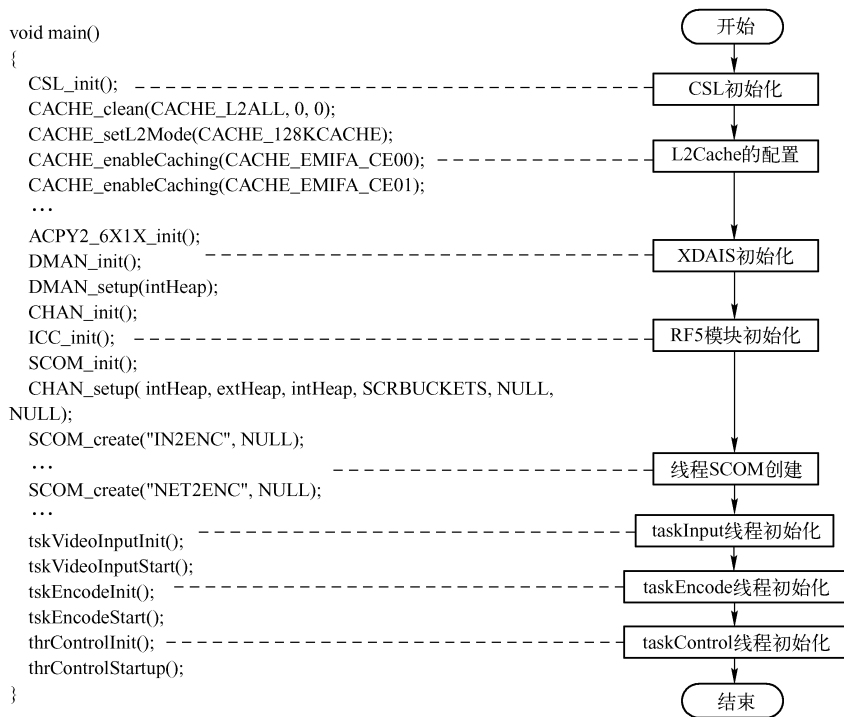


图 2.40 main 函数流程

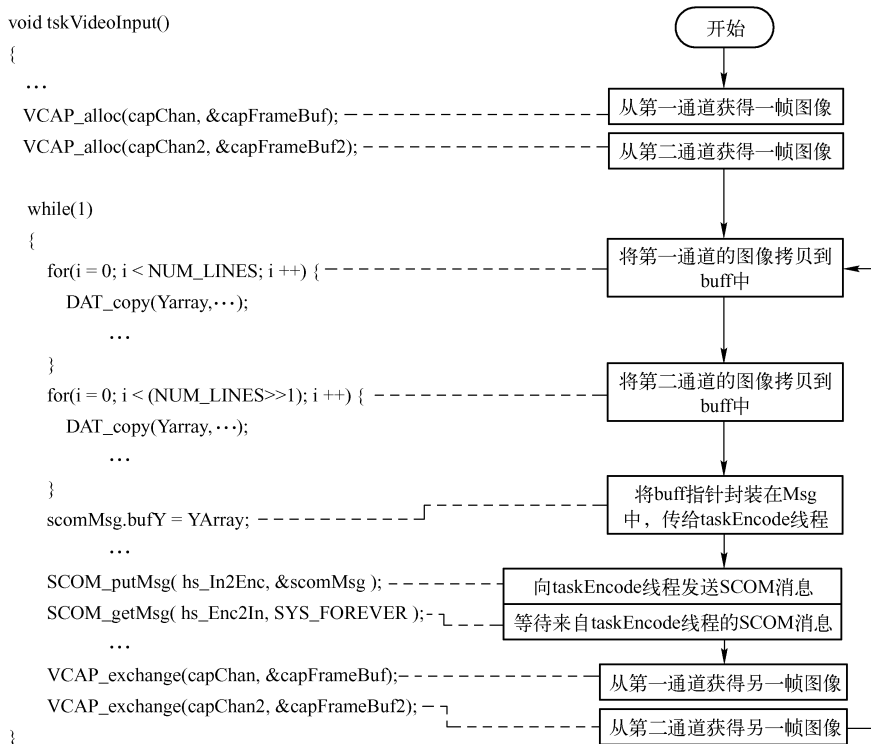


图 2.41 taskInput 线程流程

taskEncode 线程在获得来自 taskInput 线程的 SCOM 消息后，便从自身的接收 SCOM 队列中取出存储视频图像 buff 的指针，并且将指针填入内核输入 ICC 对象中。随后通道（channel）通过 ICELL 对象函数，调用 JPEG 算法完成对通道输入视频图像的编码。编码后的 JPEG 图像存储在内核输出 ICC 对象所指向的数据 buff 中，并且将这一指针随同一个 SCOM 消息发送给 taskNetwork 线程的接收 SCOM 队列，通知 taskNetwork 线程一帧 JPEG 图像已经生成，可以发送给用户了。在发送 SCOM 消息给 taskNetwork 线程之前，taskEncode 线程还会发送一个 SCOM 消息给 taskInput 线程，通知 taskInput 线程一帧图像已经处理完毕，可以接收下一帧图像了。当完成一帧图像的 JPEG 编码压缩后，taskEncode 线程将由运行态转为挂起态，等待来自 taskNetwork 线程和 taskInput 线程的 SCOM 消息将其激活。具体的软件流程如图 2.42 所示。

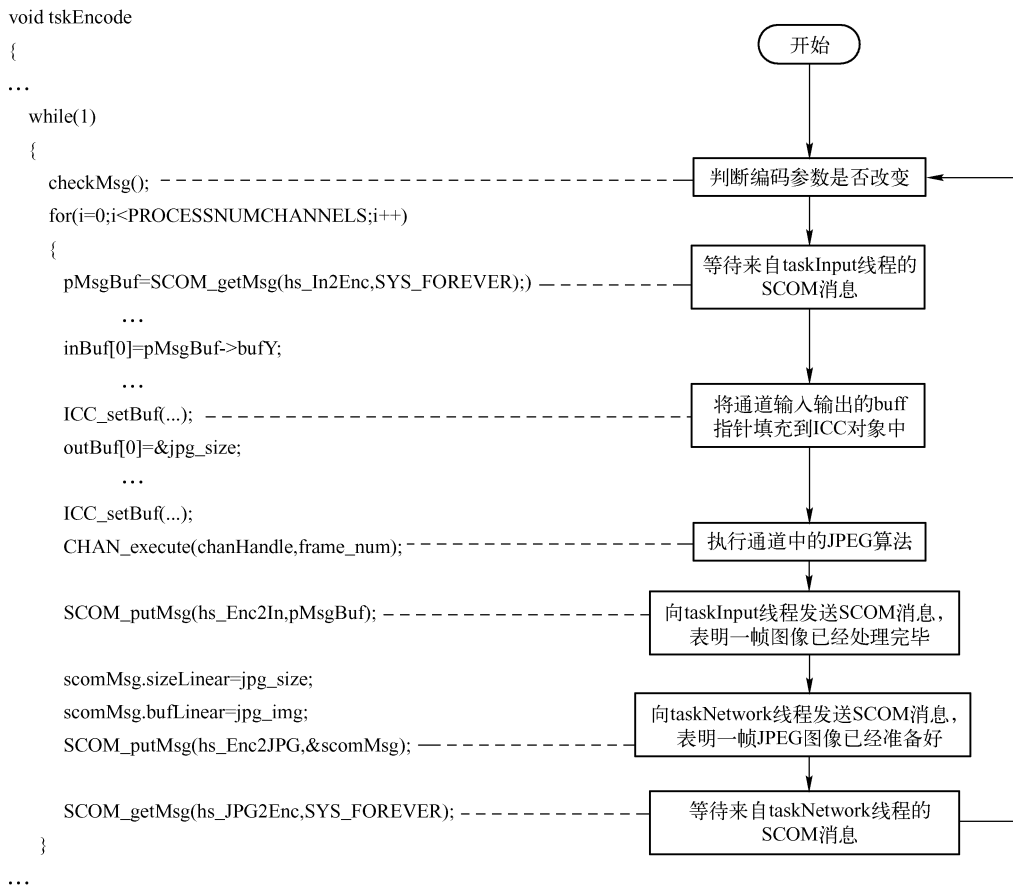


图 2.42 taskEncode 线程流程

当 taskEncode 线程处于挂起态的时候，taskInput 线程因为先于 taskNetwork 线程获得 taskEncode 线程的 SCOM 消息，taskInput 线程首先会由挂起态转化为运行态。但是当它从驱动程序获得一帧视频图像，并且发送 SCOM 消息给 taskEncode 线程后，taskInput 线程也由运行态转化为挂起态。结果就是 taskEncode 线程在等待 taskNetwork 线程的 SCOM 消息，而 taskInput 线程在等待 taskEncode 线程的 SCOM 消息。所以，taskNetwork 获得了系统的资源转化为运行态。

网络调度线程 `taskNetwork` 在初始化和配置的时候, 创建和声明了一个 `http` 网页文件 (内嵌了一个用于 `JPEG` 图像显示的 `Java Applet` 程序和一个用于控制 `JPEG` 图像压缩比的 `CGI` 程序), 并将这个文件存储在协议栈的 `EFS` 上。此外, 在网络调度线程 `taskNetwork` 还通过 `NC_NetStart` 函数创建了一个专门用于网络数据发送线程 `tskNetworkTx` 的线程。在 `tskNetworkTx` 线程中首先会创建一个基于 `TCP` 协议的套接字 (`Socket`), 然后便会侦听套接字 (`Socket`) 上是否有客户端的连接请求, 如果有, 则把创建的 `http` 网页文件通过发送函数 `Send()` 下载到客户端运行。如果没有, 则会销毁这个 `http` 网页文件, 回收内存资源。最后, `taskNetwork` 线程在转化为挂起态之前, 会发送一个 `SCOM` 消息给 `taskEncode` 线程, 通知它一帧 `JPEG` 图像已经处理完毕, 可以接收下一帧 `JPEG` 图像了。具体的软件流程如图 2.43 所示。

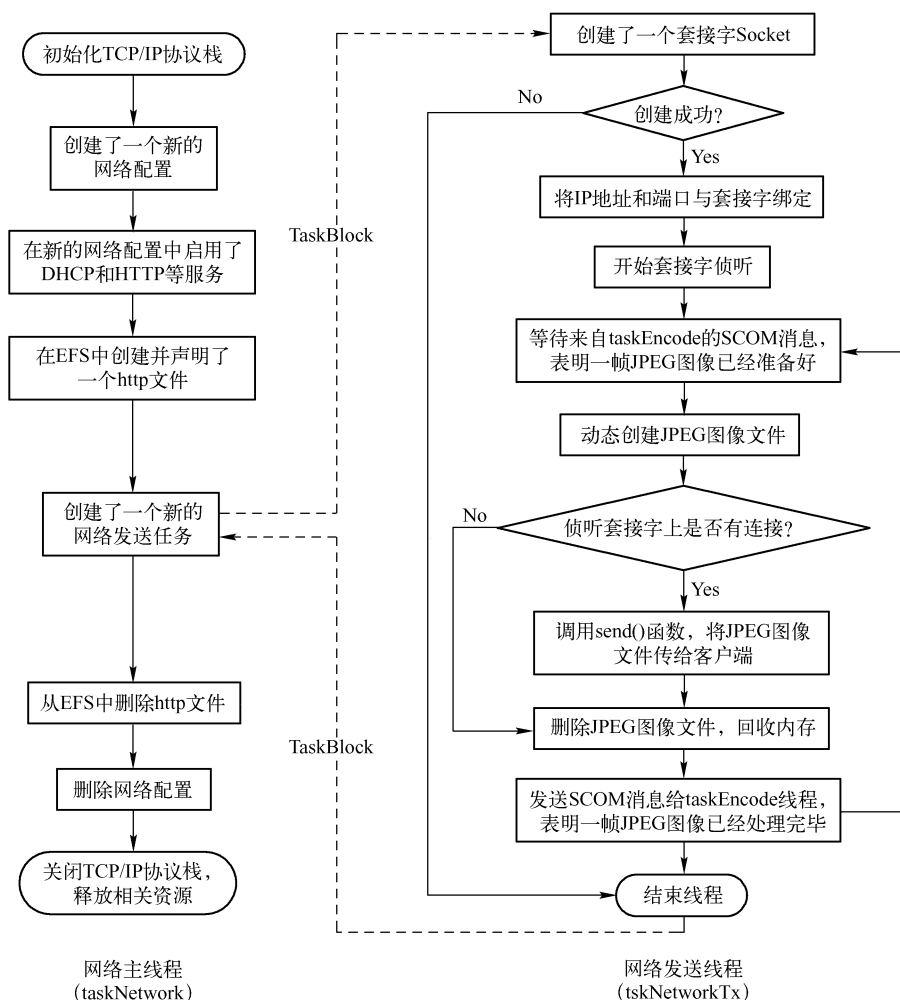


图 2.43 网络线程流程

系统中还有一个 `taskControl` 线程, 它的优先级最低。`taskControl` 线程每隔一段时间便会检查 `JPEG` 压缩比全局变量被客户通过 `CGI` 程序所更改, 如果有变化就把新的全局变量值封装在邮箱 (`MBX`) 中传递给 `taskEncode` 线程。软件流程如图 2.44 所示。

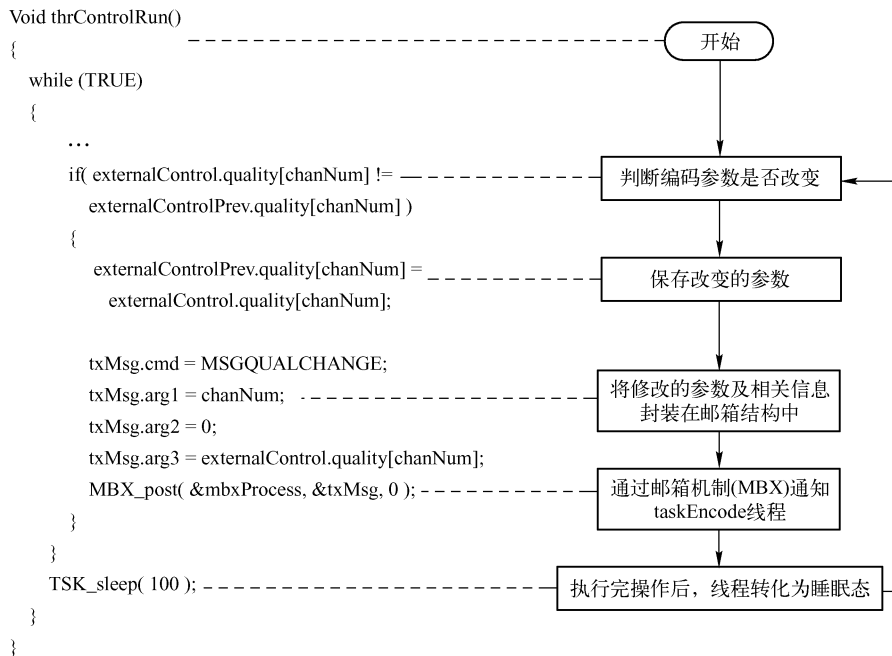


图 2.44 taskControl 线程流程

## 参考文献

- [1] TMS320 DSP Algorithm Standard Rules and Guidelines User's Guide. Texas Instruments Incorporated, February 2007.
- [2] TMS320 DSP Algorithm Standard Developer's Guide. Texas Instruments Incorporated, October 2002.
- [3] TMS320 DSP Algorithm Standard API Reference. Texas Instruments Incorporated, February 2007.
- [4] Steve Blonstein. Reference Frameworks for eXpressDSP Software: A White Paper. Texas Instruments Incorporated, December 2002.
- [5] Alan Campbell, Davor Magdic, Todd Mullanix, Vincent Wan. Reference Frameworks for eXpressDSP Software: API Reference. Texas Instruments Incorporated, April 2003.
- [6] Alan Campbell, Yvonne DeGraw. Reference Frameworks for eXpressDSP Software: RF1, A Compact Static System. Texas Instruments Incorporated, May 2003.
- [7] Alan Campbell. Achieving Zero Overhead With the TMS320 DSP Algorithm Standard IALG Interface. Texas Instruments Incorporated, November 2000.
- [8] Davor Magdic, Alan Campbell, Yvonne DeGraw. Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi - Channel, Multi - Algorithm, Static System. Texas Instruments Incorporated, April 2003.
- [9] Todd Mullanix, Davor Magdic, Vincent Wan, etc. Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High - Density System. Texas Instruments Incorporated, April 2003.
- [10] Texas Instruments. TMS320DM642 Video/Imaging Fixed - Point Digital Signal Processor. Texas Instruments Incorporated, October 2010.
- [11] Texas Instruments. TMS320C64x DSP Video Port/VCXO Interpolated Control (VIC) Port Reference Guide. Texas Instruments Incorporated, November 2010.
- [12] Murat Karaorman, Gunjan Dang, Ezell Young. Using DMA with Framework Components for 'C64x+. Tex-



---

as Instruments Incorporated, October 2007.

- [13] Texas Instruments. xDAIS – DM (Digital Media) User Guide. Texas Instruments Incorporated, January 2007.
- [14] Abhinaba Basu. Using External References in Algorithms Compliant with the TMS320 DSP Algorithm Standard. Texas Instruments Incorporated, August 2003.
- [15] Texas Instruments. Framework Components DMAN3/ACPY3 Users Guide. Texas Instruments Incorporated, July 2014.

## 第3章 多核嵌入式软件开发

随着嵌入式系统越来越复杂，ARM + DSP 多处理器或多核的架构成为嵌入式系统的主流配置；网络连接也成为必备功能；同时各种外设层出不穷，这些都对软件开发带来挑战。本章介绍 TI 为多核开发提供的相关支持。首先讨论 DSP/BIOS 实时内核，然后介绍网络开发包 NDK 以及设备驱动开发包 DDK，最后介绍用于多核间通信的基础软件 DSP/BIOS LINK。

### 3.1 DSP/BIOS 实时内核

DSP/BIOS 是一个嵌入式实时操作系统内核，能够大大方便用户编写多任务应用程序。DSP/BIOS 包括任务的调度、任务间的同步和通信、内存管理、实时时钟管理、中断服务管理等组件。用户使用 DSP/BIOS 可以编写复杂的多线程/多任务程序，并且会更合理地使用 CPU 和内存资源。

需要说明的是，DSP/BIOS 的版本目前最高是 5.41，后面的版本改为 SYS/BIOS。SYS/BIOS 主要支持 C6000 系列 DSP 芯片，其使用与 DSP/BIOS 非常相似。SYS/BIOS 还可以用于 ARM 核以及同构或异构的多核芯片。

#### 3.1.1 DSP/BIOS 简介

DSP/BIOS 是一个可裁减实时内核，提供了抢占式的多任务调度，以及对硬件的及时反应，实时分析和配置工具等。同时提供丰富的 API 接口，易于使用。DSP/BIOS 内核有效地扩展了 DSP 指令集，构成了实时 DSP 应用的底层体系架构。

由于 DSP/BIOS 内核包含了实现多种运行线程模块和设备无关的输入/输出组件，开发者可以开发各种类型的应用程序，包含简单的单通道信号处理系统和复杂的多频率和多通道系统。

DSP/BIOS 内核既可以帮助开发者管理系统资源，也是构建 DSP 应用程序的基础架构。这些内核服务调整和优化了规模和性能，并且在 TMS320C5000 和 TMS320C6000 的应用程序上可用。此外，DSP/BIOS 内核提供了硬件抽象功能，使得应用程序易于迁移到其他 TMS320DSP 上。

在应用程序的开发过程中，如何使用到 DSP/BIOS 内核呢？从设计到部署，DSP/BIOS 内核都为应用程序的开发提供必要的工具，为开发者构建和部署应用程序提供稳定的基础设施。

在应用程序的概念设计阶段，开发者根据产品需求提出多种可能的方案并进行评估。在目标平台可用前，开发者开发出原型程序，并使用 TMS320 评估模块（EVM）或者 DSK（DSP starter kit）进行评估。在这个阶段，构建原型程序是快速确定和解决技术难点的关键。

为了简单快速地开发应用程序，开发者可以结合 DSP/BIOS 内核和目标硬件抽象服务，快速构建和测试其应用程序的逻辑模块。

开发者使用 CCS 中的 DSP/BIOS 配置工具从可测量的 DSP/BIOS 内核中选择和配置应用程序运行时所需的支持组件。设计者使用这些组件构建包含运行线程、I/O 及其之间交互的应用程序框架，完成对应用程序的逻辑开发和验证。当目标平台可用时，开发者可以生成目标系统内存的逻辑视图，从而简化对目标平台的移植。

配置工具生成的 DSP/BIOS 内核对象包含了实时分析所需的内部指令。通过使用实时分析工具，设计者能快速地度量使用 DSP/BIOS 组件构建的应用程序框架所需的消耗。一旦开发者验证了应用程序的逻辑，它们就可以加入算法模块。DSP/BIOS 内核里的 RTA 组件也为算法运行时的操作模块提供了可视化开发方法。

开发者可以通过定制测试向量实现自己的算法，可以支持监测与通知、数据模拟和实时数据捕获。与实时数据交换（RTDX）模块相结合，开发者可以根据不断更新参数和监视应用程序在 DSP 上运行而实时产生的结果来调整他们的算法。

最后，在集成阶段，实时交互过程中经常会出现错误或故障。由于这些错误或故障通常不是周期性出现，发生的频率也很低，所以很难被发现。但是，结合开发者设计的自定义测试向量，DSP/BIOS 内核中的 RTA 模块为这些事件提供了可视化跟踪。可视化能尽可能地帮助开发者隔离和修正集成过程中出现的困难和问题。

使用 DSP/BIOS 开发 DSP 软件有两个重要特点：第一，所有与硬件有关的操作都必须借助 DSP/BIOS 本身提供的函数完成，开发者应避免直接控制硬件资源，如定时器、DMA 控制器、串口、中断等；第二，使用 DSP/BIOS 后，由 DSP/BIOS 程序控制 DSP，用户的应用程序建立在 DSP/BIOS 基础之上。用户程序也不再是按编写的次序顺序执行，而是在 DSP/BIOS 的调度下按任务、中断的优先级排队等待执行。

一个使用 DSP/BIOS 开发的应用程序，主要是通过调用一系列的 DSP/BIOS 实时库中的 API（应用编程接口）函数来实现。这些 API 函数的目的是，在目标系统的嵌入式程序中，包括在实时、I/O 模块、软件中断管理、时钟管理等情况下，捕获信息，进行操作。DSP/BIOS 的 API 分成多个模块。由于应用时配置的不同，DSP/BIOS API 的代码长度，从 200 至 2 000 字节不等。就 C 程序而言，应用程序的头文件定义了 API。就汇编语言的优化而言，程序可以使用 DSP/BIOS 定义的宏。

开发者使用 DSP/BIOS 配置工具为自己的应用程序选择合适的 DSP/BIOS 模块，他们也可以利用这些模块来声明和配置对象。这可以帮助预生成节省内存的 DSP/BIOS 对象，也可以对配置参数进行预验证。DSP/BIOS 配置工具可以和 CCS 紧密集成，将在第 5 章介绍。

### 3.1.2 DSP/BIOS 内核

DSP/BIOS 内核模块见表 3.1。通常情况下，单个 DSP/BIOS 模块管理一个相关类别对象的一个或多个实例，这些对象有时指向内核对象，并根据全局参数的取值控制它们的行为。开发者可以使用 DSP/BIOS 配置工具静态声明和配置这些对象，也可以在应用程序中动态声明和配置它们。

表 3.1 DSP/BIOS 功能模块

属性	模块名称	模 块 说 明	对象创建方式		支持的语言	
			静态	动态	C	汇编
实时分析和数据捕获						
事件日志	LOG	消息日志管理器	×		×	×
统计累加	STS	统计累加器	×		×	×
跟踪控制	TRC	跟踪管理器	×		×	×
文件流	HST	主机 I/O 管理器	×		×	×
实时数据交换	RTDX	目标 – 主机通信管理器	×		×	×
硬 件 抽 取						
片上计时器	CLK	系统时钟管理器	×		×	×
硬件中断	HWI	硬件中断管理器	×			×
静态内存管理	MEM +	内存段管理器	×			
动态内存管理	MEM *	内存段管理器		×	×	
设备无关 I/O						
数据管道	PIP	数据管道管理器	×		×	×
数据流	SIO/DEV	流 I/O 管理器	×	×	×	
执行线程管理						
软件中断	SWI	软件中断管理器	×	×	×	×
周期函数	PRD	周期函数管理器	×		×	×
任务	TSK	多任务管理器	×	×	×	
空转循环	IDL	空转函数循环管理器	×		×	×
内部线程通信和同步						
信号	SEM	信号管理器	×	×	×	
资源锁	LCK	资源锁管理器	×	×	×	
邮箱	MBX	邮箱管理器	×	×	×	
队列	QUE	队列管理器	×	×	×	
其他服务						
原子函数（优化和不可被抢占）	ATM	汇编语言写的原子函数	N/A	N/A	×	
错误处理和程序中止	SYS	系统服务管理器	N/A	N/A	×	
片支持库（CSL）						
直接内存访问	DMA		×	×	×	
改进的直接内存访问	EDMA	仅供 TMS320C6x	×	×	×	
外部内存	EMIF	仅供 TMS320C6x 和 TMS320C55x	×	×	×	
多通道缓冲区的串行端口	McBSP		×	×	×	
计时器	TIMER	计时器设备	×	×	×	
伸缩总线	XBUS	仅供 TMS320C6x	×	×	×	
指令高速缓冲存储器	CACHE	仅供 TMS320C55x	×	×	×	
通用目标输入/输出	GPIO	仅供 TMS320C5x	×	×	×	
时钟生成器	PLL	仅供 TMS320C5x	×	×	×	
监视狗计时设备	WDTIMER	仅供 TMS320C54x	×	×	×	

### 1) 用 DSP/BIOS 执行线程结构化应用程序

对于互不相关的执行路径组成的应用程序,开发者可以给出这些路径的结构和次序。DSP/BIOS 执行线程是一些互不相关的执行路径,每个线程执行一些互不关联的 DSP 指令流。一个执行线程是一个单一的控制点,由中断处理程序 (ISR)、子程序或者函数调用构成。例如,一个硬件中断是一个线程,一旦触发,它就执行中断处理程序。

通过利用执行线程组织 DSP 应用程序,开发者可以结构化他们的应用程序,为每个线程分配相应的优先级。通过优先级,多线程应用程序可以在单处理器系统上运行,且高优先级的线程比低优先级的线程优先执行。DSP/BIOS 内核提供了 30 个优先级,可以分配给 4 类不同的执行线程。它还提供了服务支持执行线程间的同步和通信。

每个线程类有不同的执行、抢占和挂起特征。既然所有线程都是完全可抢占的,开发者就可以轻松地在基于 DSP/BIOS 的应用程序中集成已有的算法,而无须修改算法源代码。此外,在硬实时线程要获取 CPU 时,抢占线程是一种简单的方法。加入新线程并不会破坏系统的正确性。硬件中断和准备线程之间所花费时间的长度不会受系统中线程数目的影响。

除了后台空转处理线程,每个类型的线程支持不同的优先级别。DSP/BIOS 内核提供不同的选择,允许开发者为自己的应用程序定义合适的线程类型,并不强制应用程序符合一个特定的模型。因为对应用程序没有完全正确的解决方案,DSP/BIOS 开发者可以灵活地组合和匹配运行库中最适合应用程序的对象。此外,DSP/BIOS 是完全可以度量的,只有被选择的模块才可以与应用程序相链接,从而减少资源需求。

#### ➤ 硬件中断

在 DSP/BIOS 内核中,硬件中断 (HWI) 模块管理了一组有限数目的响应特定硬件中断的对象,这些对象被底层的 DSP 平台所辨识。HWI 模块提供了支持来运行与这些对象相关联的中断处理程序,以调度执行 DSP/BIOS 软件中断或者同步任务。HWI 模块还提供运行时服务来启动或者禁止执行硬件中断。在 DSP/BIOS 配置工具中,开发者可以使用 HWI 管理器将中断处理程序和硬件中断建立映射,在内存中定位中断处理表和 HWI 调度器。在 DSP/BIOS 应用中,开发者能用汇编语言或汇编和 C 的混合语言编写中断处理程序 (ISR)。使用 HWI 管理器的一个主要好处就是设备抽象。HWI 模块允许开发者使用逻辑服务进行配置、管理和使用硬件中断。当应用程序需要移植到其他设备,开发者只需将逻辑视图重新制定给新的物理设备。

#### ➤ 软件中断

顾名思义,这个执行模块和硬件的中断处理程序类似,只是软件中断并不和物理的 DSP 设备相关联,他们只是软件的实例化。和硬件的 ISR 一样,软件中断也以连续运行模式执行,它们共享抢占属性。DSP/BIOS 软件中断是基于优先级的,支持 14 种优先级。软件中断 (SWI) 模块管理 DSP/BIOS 软件中断。软件中断只能被较高优先级的软件中断或者硬件中断所抢占。在相同优先级上,应用程序可能有多个软件中断。这时,处于相同优先级的软件中断按照先来先服务的规则执行。当外部事件触发了硬件中断,程序就会调用 SWI 函数触发软件中断。程序还可以通过调用 DSP/BIOS API 来触发软件中断。这些调用语句可以被嵌入在目标程序的执行线程中。目前的 TMS320 设备上,每个中断程序调用 SWI\_post() 时,DSP/BIOS 的负载至多是 1 毫秒,这对于处理周期为 1kHz 的应用程序几乎没有影响。

正因为上下文切换的低负载,DSP/BIOS 软件中断对象成为了结构化程序的一个理想机制,使得程序能实现在不同频率上运行多种算法,例如,一个能编码语音、识别音调和消除

回声的无线电通信应用程序，可以使用范围在 1 ~ 20ms 内有不同周期的多帧来处理一个 8kHz 输入流。使用单调比率调度技术，将较少限制的算法和较高优先级的软件中断相绑定，可以保证其他的独立实时线程可以按序交叉地运行，从而每个线程都能被分配到处理器周期。因为软件中断处理程序在严格的优先级原理上来抢占处理器周期运行，这使得单调比率调度技术是可行的。

### ➤ 周期性函数

DSP/BIOS 内核提出一类特殊的软件中断，这类软件中断由一个周期时钟发出。该时钟用于调度周期函数或按周期率发生的活动。DSP/BIOS 中使用 PRD 模块管理这些函数。DSP/BIOS 提供了一个系统时钟，即一个通过调用 PRD\_tick() 而实现的 32 位计数器。计时器中断通过调用 PRD\_tick() 来驱动这个系统时钟。但是，其他的周期事件，如数据时钟，也能调用 PRD\_tick()。PRD 管理器允许开发者调度 PRD 软件中断线程来运行不同比率上的函数。开发者可以创建任意多的 PRD 对象，每个 PRD 对象对应一个不同的周期。然而，因为所有的 PRD 对象由同一个系统时钟所驱动，比率是系统时钟或者 PRD\_tick() 的乘积。PRD 对象包含了一个函数、两个参数和一个指定成功调用之间时间的周期。有的应用程序需要有一个函数，该函数在一次时间延迟后只执行一次，DSP/BIOS 周期函数支持单循环或一次轮转模式。因为 PRD 对象是由软件中断激发运行的，所以它们与软件中断共享相同的堆栈。

### ➤ 同步任务

与前面介绍的硬件中断和软件中断模型不同，DSP/BIOS 内核中的任务既能挂起也能被抢占。任务从运行直到完成期间，都可能被抢占或挂起。同步任务运行在比软件中断低的优先级上，但是比空闲后台线程的优先级高。DSP/BIOS 任务是基于优先级的，支持 15 个优先级级别，和一个挂起状态。任务可以被硬件中断和软件中断抢占，也可以被更高优先级的任务抢占。在 DSP/BIOS 中，TSK 模块管理和调度任务。

一旦任务被挂起，线程的执行也就被挂起（或阻塞），直到某个资源可用或者某个事件发生。任务可以被自己阻塞，也可以被其他任务阻塞，有以下几种方式：

- 主动地让位 (TSK\_yield());
- 休眠一段时间 (TSK\_sleep());
- 等待某个资源变成可用或者发生某个事件，即使用信号进行同步（如 SEM\_pend() 或 MBX\_pend()）。

**注意：**硬件中断和软件中断不能休眠或阻塞。没有被抢占的情况下，它们总是运行完成。

DSP/BIOS 提供了类似的内核元素实现传统的并发处理。在传统的并发系统设计中，多线程应用程序由一组基本成分构成，它们与任务、信号、邮箱和消息队列类似。熟悉 Vx-Works、PSOS 和 Nucleus 的开发者在 DSP/BIOS 内核中能发现相似的组件。

### ➤ 后台空闲处理

在 DSP/BIOS 应用程序中，后台空闲线程在最低的优先级运行。在没有更高优先级的线程（如中断或任务）需要运行时，这个线程就不断运行。有时，由互不关联的执行线程组织的应用程序中，一个线程可以是一个只在没有其他线程需要运行时才应该运行的操作（即后台运行）。在轮流检测非实时 I/O 设备或不能产生中断通信端口、监视系统状态和其他开发者不希望影响实时应用程序的操作时，这种线程就非常有用。事实上，CCS 的 RTA 插件和目标应用程序之间的通信就是在后台循环运行的，这样可以保证主机链接不会干扰实时应用。DSP/BIOS 配置工具中的 IDL 管理器允许开发者在空闲循环中添加函数。在一个时



刻, 这个空闲循环会按照排好的顺序调用每个函数, 每个函数也可以运行直到完成。这个过程周期性地重复。如果希望实现最小功率的操作, 开发者可以在 DSP/BIOS 内核和后台空闲循环处理间权衡选择。内核允许开发者响应最高级别线程的同时, 也能在没有线程运行时空闲运转处理器。在空闲循环中, DSP/BIOS 内核知道能空闲运转处理器。开发者能决定, 是空闲运转和还是停止处理器运转, 或执行其他低功率模式。

## 2) DSP/BIOS 实时分析 (RTA)

如图 3.1 所示, DSP/BIOS 实时分析特征为开发和集成提供了可视化来帮助开发者探测、跟踪和监视执行中的 DSP 应用程序。事实上, 这些工具的功能就相当于由调试器部署的物理 JTAG 连接器, 利用这个连接器可以在目标程序和主机之间建立低速实时的通信链接。

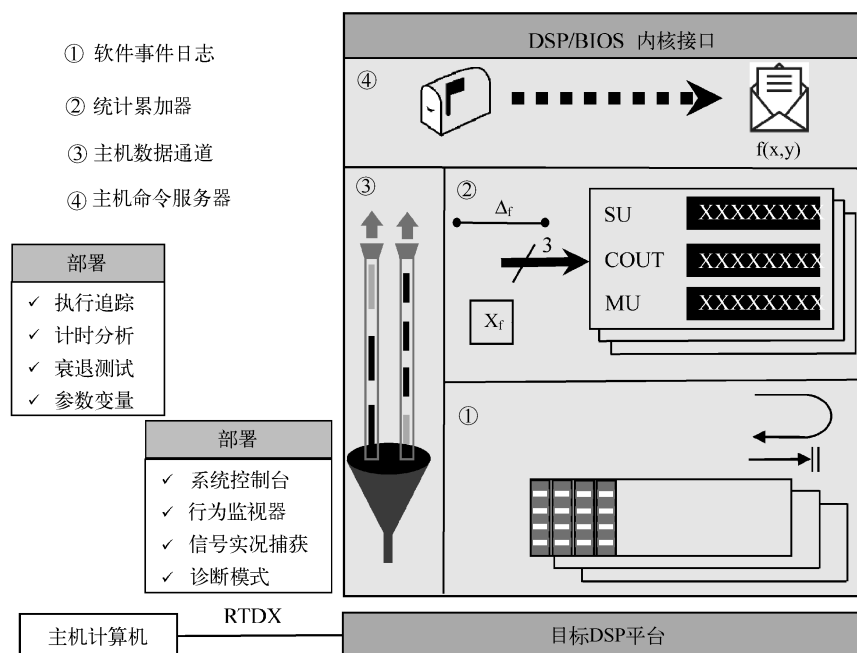


图 3.1 DSP/BIOS 的实时捕获和分析

DSP/BIOS 的 RTA 作为 DSP/BIOS 内核存在与目标系统中。作为为应用程序提供的额外运行时服务, 通过物理链接, DSP/BIOS 内核为于主机的实时通信提供支持。由于可以利用 DSP/BIOS API 简单结构化应用程序, 并能为多任务和 I/O 支持静态创建对象, 开发者能在目标程序中自动地使用 CCS IDE 内部的可视化分析工具来捕获和上载实时信息。辅助的 API 和对象也允许在目标程序的控制下直接捕获信息。根据主机方工具的设置, DSP/BIOS 内核为实时程序分析提供下列主要能力。

- **消息事件日志**——能按照时间顺序显示写入独立于实时线程的内核日志对象的事件, 并能跟踪程序控制流。目标程序显式调用 DSP/BIOS API、线程被准备好、被调度和被终止时底层内核会隐式调用 API 时, 都会记录相应的事件。
- **统计累加器**——能显示内核累加器对象收集的统计数据, 这些数据反映了各种动态程序元素, 从简单的计数器和随时间变化的数值到独立线程所消耗的处理时间间隔。通过显式调用 DSP/BIOS API 或调度线程为执行 I/O 操作而隐式调用, 目标程序得到这些统计累加数据。

- 主机数据通道——能将内核 I/O 对象与主机文件绑定，这些主机文件为目标程序提供标注数据流以定时测试算法。在不工作时记录和捕获的由内核 I/O 对象管理的其他实时目标数据流也要与主机文件绑定，以支持并发分析。
- 主机命令服务器——能控制目标程序中的实时跟踪和统计累加。可以有效地让开发者控制实时程序执行过程中的可视化程度。

在软件开发过程中，与 CCS 标准调试器一起起作用，在目标程序执行过程中，在准确的时间间隔中，DSP/BIOS 实时分析工具为目标程序的行为提供了重要的可视化。在这些地方，调试器较难甚至不能进行调试。即使在调试器中止程序和控制目标程序后，DSP/BIOS 已经捕获的信息能提供在当前执行点之前的关于事件序列的宝贵信息。

在软件开发周期的后期阶段，对于处理由程序组件之间时间相关的交互所引起的更诡异的问题，普通的调试器变得无效。DSP/BIOS 实时分析工具包含了一个可扩展的功能，相当于软件形式的硬件逻辑分析器。

在软件开发结束后，DSP/BIOS 的这个功能变得更显著。在测试生成和域诊断工具里，嵌入的 DSP/BIOS 内核及其结合的主机分析工具提供了必要的基础。通过已有的 JTAG 基础设施，这些工具将捕获运行产品系统中应用程序之间的交互。

### 3) 实时数据交换 (RTDX)

在不妨碍目标应用程序的情况下，实时数据交换帮助系统开发者在主机计算机和 DSP 设备之间传递数据。这种双向通信通道是为主机收集数据而提供的，也是主机和正在运行的 DSP 应用程序之间交互所需要的。从目标程序收集数据可以用于在主机上分析和建立可视化。在不停止应用程序的前提下，也可以使用主机工具调整应用程序的参数。RTDX 也使主机系统能向 DSP 应用程序和算法提供模拟数据。

RTDX 包含了目标程序和主机的组件。在目标 DSP 上运行 RTDX 软件。DSP 应用程序调用这个软件的 API 来传递数据。这个软件使用基于扫描的仿真器，通过 JTAG 接口在主机平台和目标程序间移动数据。当 DSP 应用程序运行的同时，数据就会被传送给主机。

在主机平台上，RTDX 主机软件与 CCS 相关联。数据可视化和分析工具通过 COM API 与 RTDX 通信，以获取目标数据并/或发送数据给 DSP 应用程序。

主机软件支持两种模式从目标应用程序接收数据：连续和非连续。在连续模式中，RTDX 主机软件简单地缓冲数据，并不写到日志文件中。在开发者希望连续获得并显示 DSP 应用程序的数据，并且不需要在日志文件中存储数据时，就应该使用连续模式。在非连续模式中，需要将数据写入主机上的日志文件中。当开发者希望捕获和记录有限数量的数据，就应该使用此模式。

### 4) 硬件抽象

DSP/BIOS 内核提供 API 来访问和配置某些硬件组件的一些独立于它们的物理实现的特性。硬件抽象 API 提供了一个简单的独立于底层设备的逻辑用户接口，来配置这些设备。抽象设备相关的组件（如使用 CLK 抽象片上计时器）和硬件中断（HWI）使移植程序变得额外简单。图 3.2 说明了 DSP/BIOS 硬件抽象服务。

DSP/BIOS 内核也支持内存管理。通过 DSP/BIOS 配置工具，开发者定义和命名物理内存段，创建逻辑内存视图。和 MEM 模块一起使用逻辑内存视图，DSP/BIOS 运行时应用程序接口能为应用程序动态分配和释放内存。

独立于设备的 I/O 应用程序接口提供了基于帧或基于块的数据传输服务，这些服务为

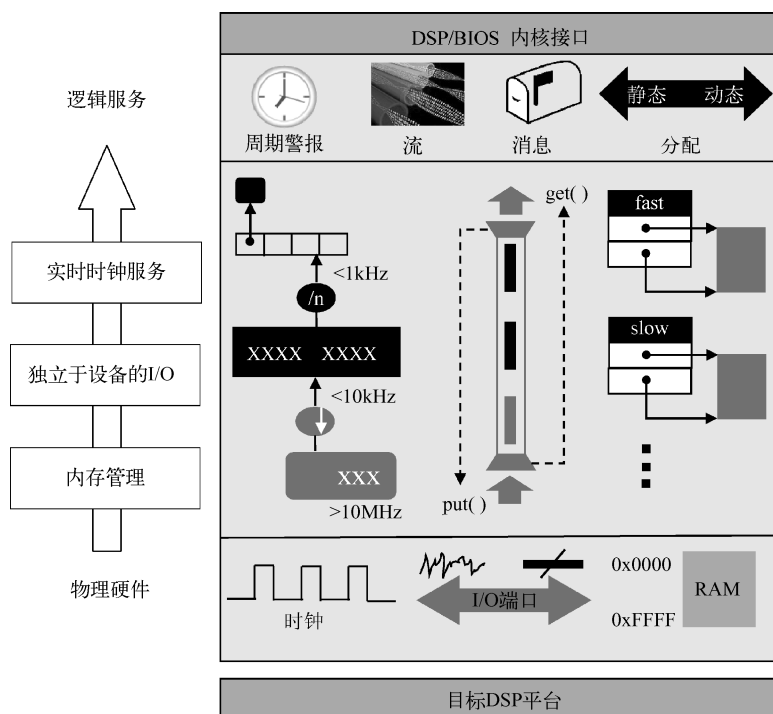


图 3.2 DSP/BIOS 的硬件抽象服务

DSP 和外围设备之间和多执行线程之间进行数据传输。典型的外设包括类似 CODEC 的 I/O 设备和主机系统。DSP/BIOS 支持数据管道和数据流。

### ➤ 实时时钟服务

CLK 模块允许开发者配置与硬件相关的组件，如片上计时器，为所有应用程序的计时器或周期函数提供时基。DSP/BIOS 配置工具提供了一个简单的逻辑接口，将片上计时器调整到期望的时基，例如 1 微秒。并且在需要的时候，可以访问物理计时器寄存器。

CLK 模块提供了实时时钟的抽象函数以两种方式来访问时钟。与统计累加器对象相结合，时钟可以用来度量传递时间，也可以为事件日志添加时间戳消息。低分辨率和高分辨率的时间都以 32 - 位数值的方式存储。低分辨率时钟根据计时器中断率计时，时钟的取值等于已经发生的计时器中断次数。高分辨率时间是计时器寄存器已经增加的次数，它是一个 DSP 时钟的直接函数。高分辨率时间对于判断 DSP 执行指令序列所花费的时间非常有用，可以使用指令周期作为度量的单位。

时钟管理器允许开发者产生任意数目的时钟函数。每当计时器中断发生时，时钟管理器就执行时钟函数。ISR 允许的 DSP/BIOS 操作也可以调用这些函数。

### ➤ 内存管理

DSP/BIOS 内核为 TMS320 平台提供内存管理。DSP/BIOS 内存区管理器允许开发者制订所需的内存段来定位 DSP/BIOS 应用程序的代码和数据区。MEME 模块还提供了一组运行时函数来分配存储区。开发者使用 DSP/BIOS 配置工具来制定内存段，形成物理系统的逻辑内存视图。在配置工具中，开发者可以在逻辑层使用逻辑内存视图编程。抽象物理内存视图可以简化应用程序在硬件平台和新的 TMS320 设备上移植的过程。

DSP/BIOS 的软件模块使用运行时 MEM 函数在运行时分配存储空间。结合配置工具，

DSP/BIOS 模块使用 MEM 在选定的内存段上分配存储区域。

### ► 独立于设备的 I/O

所有 DSP 应用程序基本都需要获取数据、处理数据和输出结果。典型的 DSP 应用程序会同时处理数据块而不是单个数据。所以，这些应用程序会从数据源移动连续的数据块，并且处理和输出结果。从概念上来讲，数据块的移动就形成了一个从数据源到接收器的单向数据流。为了能异步管理多数据帧，这些数据流允许在不同频率上进行 I/O 和处理，即 DSP 处理前一个已装载的缓冲区的同时设备能往另一个缓冲区中填充数据。

DSP/BIOS 内核采用两种基本机制来提供移动数据块或数据帧的服务：数据流和管道。DSP/BIOS 使用缓冲区传输机制交换地址指针，无须拷贝就能转换缓冲区，并能减小实际移动的数据量。

数据管道是简单的全局组件，能在读写线程间传递基于帧的数据。数据管道是小型高效的，在设计时是静态绑定的，可以优化性能和减小负载。

数据流也为应用程序提供独立于设备的、异步的、基于帧的传输机制。数据流可以像数据管道那样静态绑定，也能在程序执行时动态绑定。数据流也提供可变缓冲区模式以支持较大范围的应用需求。为了能具有可变性，数据流依靠一个或多个底层设备驱动器。设备驱动器压缩了与设备相关的属性和函数。设备驱动器也能使用堆栈设备驱动器机制来执行数据传输上的其他操作。这类设备驱动器能完成管道处理操作，如数据类型转换、调整或过滤数据通道。

### ► 数据管道

数据管道提供了一个代码高效的全局数据传输机制，实现中断服务处理程序和延迟过程（如软件中断或任务）之间的通信（如图 3.3 所示）。数据管道是无向的，它们传输数据缓冲区或两个 DSP/BIOS 执行线程间的帧。在 DSP/BIOS 内核中，数据管道由 PIP 模块管理。

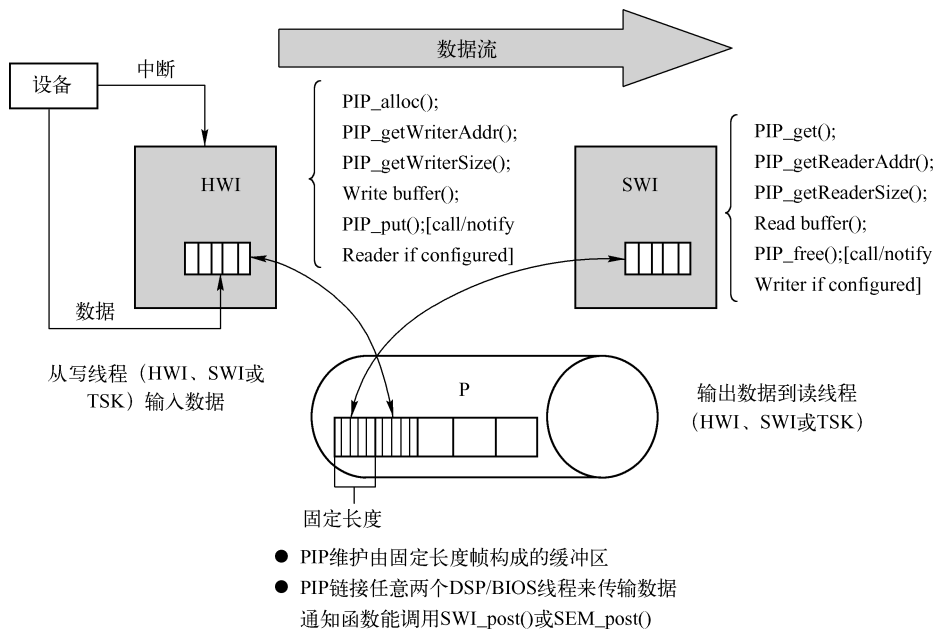


图 3.3 DSP/BIOS 的数据管道

数据管道支持两种同步数据传输模式：poll 函数和 callback 函数。执行线程能投票选择数据管道来同步数据传输；或为事件驱动的应用程序，数据管道能调用通知函数（callback

函数) 通知一个可用的缓冲区。例如, 硬件 ISR 用输入的数据填充一个缓冲, 并调用 PIP\_put() 将缓冲区放回数据管道。数据管道管理器会调用通知函数, 通知管道末端链接的线程有一个缓冲区可用。

数据管道有各自的内存池, 每个内存池是由一定数目的固定长度的缓冲区(或帧)构成的。数据管道在读出器和数据管道或数据管道和写入器之间交换这些数据帧。如果应用程序需要向另一个数据管道传输缓冲区, 应用程序就需要拷贝数据到其他数据管道中。在设计时, 需要使用 DSP/BIOS 配置工具静态声明和绑定数据管道。

#### ➤ 数据流

数据流既支持静态声明和绑定, 也支持运行时创建和绑定。数据流为应用程序提供了比数据管道更高的灵活性和更强的结构化。对数据流的操作包括开关流设备的操作、启动、停止和刷新数据流, 以及提供控制的操作。在缓冲区模式和内存管理两方面, 数据流都提供可变性。一个缓冲区可以归某个数据流私有, 也可以公有。

在 DSP/BIOS 内核, SIO 模块管理应用层次的数据流。根据 SIO 的独立于设备的抽象功能, 所有设备对于应用程序而言都是一样的。管理设备相关驱动器的 DEV 模块被 SIO 补充并与 SIO 进行交互。

#### ➤ SIO 设备驱动器

底层设备驱动器能完成各种功能。DSP/BIOS 支持两类设备驱动器: 终端和堆栈。

终端驱动器执行与外设相关联的典型 I/O 操作, 如 CODECS。设备驱动器经常包含中断服务程序 (ISR) 来与外设进行交互。但是, 其他的 I/O 操作 (如 polling) 也同样是被允许的。

堆栈驱动器是一类特殊的驱动器, 执行内嵌管道处理过程。堆栈设备驱动器的典型应用是实现调整或过滤操作。这些驱动器能支持各种大小的数据和缓冲区, 缓冲区的长度也是可以变化的。这对于不同数据格式转换是非常有用的, 如固定指针到浮动指针或 14 位音频到 16 位数据。既然压缩和复用这些操作是有益的, 堆栈设备驱动器就会考虑实现它们。

#### ➤ 片级支持库 (CSL)

片级支持库 (CSL) 提供了配置和管理片上外设的 C 语言接口。它由几个独立的模块组成库文件。CSL 模块是顶层应用程序接口模块。CSL 模块的基本目的是初始化这个软件库。在调用其他 CSL 应用程序接口函数前, 在程序开始, 必须调用 CSL\_init() 函数一次。

开发者可以使用配置工具对外设进行编程, 如直接内存访问 (DMA)、多通道缓冲串行端口 (McBSP) 等。

### 3.1.3 DSP/BIOS 多线程程序设计

传统的 DSP 应用程序是非常简单的, 通常是使用一个单一程序循环管理所需的处理。后来, DSP 应用程序需要并发处理, 因为应用程序需要 DSP 实现更多的功能。现在, 应用程序通常需要 DSP 同时在不同的速率上处理几个任务。此外, 使用传统的程序循环范式构造现代的 DSP 应用程序是非常困难的, 这样的应用程序难以维护, 更难以扩展。使用 DSP/BIOS 实时内核为开发者提供了构建现代的应用程序的基础, 无论是简单的应用程序还是复杂的多线程、多速率的应用程序。

对于不熟悉实时多线程内核的开发者, 使用 DSP/BIOS 内核服务架构应用程序能使你构



建结构化的灵活的应用程序。当开始熟悉 DSP/BIOS 内核及其性能后，会发现开发、设计、调试、维护和扩展应用程序很简单。对于有经验的开发者，DSP/BIOS 内核实现了嵌入式系统中发大部分的传统内核服务，你还能发现 DSP/BIOS 内核服务是高效的、可升级的，且易于实现的。

典型的嵌入式 DSP 系统所需的软件分为两种组件：应用程序和系统软件，系统软件负责管理应用程序所需的系统资源。典型的嵌入式 DSP 系统包含了 DSP 处理器、内存、系统计时器或时钟和 I/O 外设。需要使用系统软件初始化和和管理硬件组件。此外，系统软件还管理系统资源和应用软件之间的访问。这对于将应用程序迁移到不同的硬件平台（如下一代设备）上是非常重要的。系统软件是应用程序和物理硬件之间的隔离层。

简单的系统中，系统软件由基本硬件初始化、外设访问函数和硬件中断处理程序（ISR）组成。复杂的系统需要实时调度 DSP 来保证正确操作。另外，由于应用程序需要对硬件资源（如 DSP、内存或 I/O）的并发访问，高效的资源管理器和调度器就变得极为重要。管理这些资源恰恰是使用 DSP/BIOS 内核的优点。

本小节描述了如何使用 DSP/BIOS 内核构造 DSP 应用程序。主要焦点是理解使用 DSP/BIOS 组件的多线程设计方法。

### 3.1.3.1 开发过程

由于 DSP/BIOS 内核是一组可扩展的组件化的系统服务集，开发者可以完全控制用到的 DSP/BIOS 组件，使得构建的应用程序只包含所选择的那些组件。

典型的应用程序使用 DSP/BIOS 内核配置系统中断向量和系统的内存视图。在开发过程中，大部分开发者利用 DSP/BIOS 内核中内嵌的实时分析特性来获取应用程序运行时行为的可视化视图。此外，应用程序可以使用 DSP/BIOS 进度器为 DSP 的处理过程安排优先级，并管理它们的运行。已存在的应用程序能使用 DSP/BIOS 内核简化移植到其他 TMS320DSP 的过程，也能使用实时分析特性更好地了解运行时的行为和性能。

下面用一个音频应用示例来描述使用 DSP/BIOS 组件开发应用程序的过程。

#### 1) 使用多线程设计

虽然 DSP/BIOS 内核提供传统的循环架构，多线程的设计具有更好的灵活性和可维护性。即使应用程序包含了 ISR 和后台操作，使用 DSP/BIOS 线程进行设计也更易于兼容新特性和移植到其他的 TMS320DSP 上。

用一组内部相关的线程组织应用程序提供了更好的结构化，并且使得应用程序易于维护和扩展。这种结构允许开发者为线程分配相应的优先级，以确保应用程序正确执行。图 3.4 给出了使用线程重组传统循环过程的例子。

图 3.4 左图中，将过程放在后台执行可以减少中断延迟，并能提高中断带宽。由于 DSP 应用程序的复杂度增长，开发者使用后台循环来执行基本的调度，实现状态机，或保证高优先级的操作能满足它们的实时时间截止需求。然而，这些设计仍然包含了 ISR 和后台处理的循环。这种设计不利于计时，难以满足添加新特性的需求变更。

图 3.4 右图中，使用 DSP/BIOS 内核，为每个独立的执行路径分配一个线程。需要为这些线程指定相应的优先级，以确保系统能正确运行。DSP/BIOS 使用的是一个固定优先级和可被抢占的调度器，它允许应用程序定义实时调度技术，如单调速率调度（RMS），这样可以保证应用程序能正确运行。



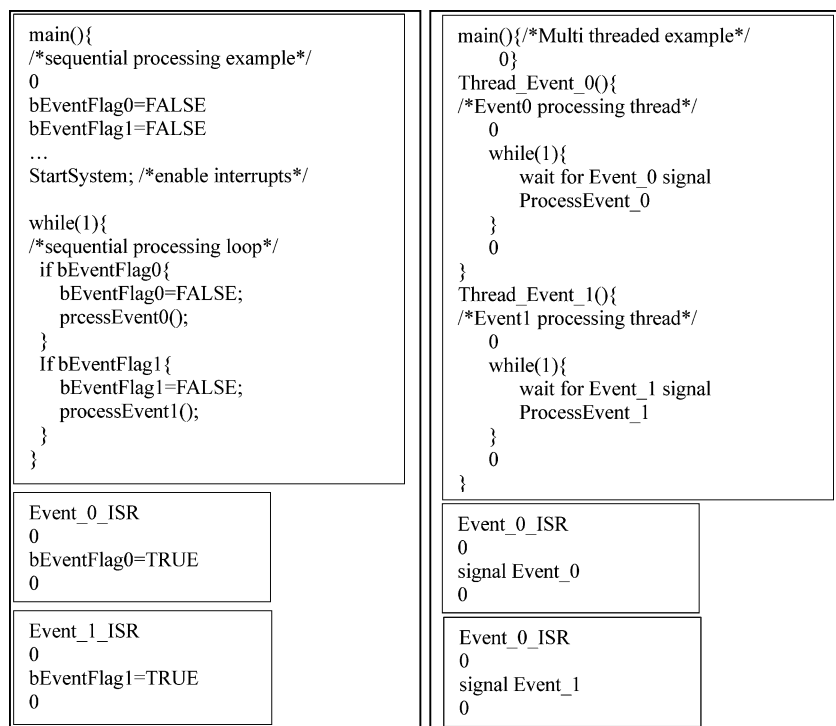


图 3.4 传统的循环过程（左）重组为多线程应用程序（右）

### 策略

在设计中，应该尽可能减少 ISR 的处理时间，时间尽量消耗在 DSP/BIOS 线程上，如软件中断（SWI）或同步任务（TSK）。可以根据操作的重要性或者实时时间截止需求为每个线程分配优先级。

#### ➤ DSP/BIOS 线程类型

如前所述，DSP/BIOS 内核支持四种基本线程。硬件中断（HWI）线程和软件中断（SWI）线程支持快速中断处理。在硬件中断中执行最少的处理，其他处理函数都采用软件中断。HWI 和 SWI 线程共享系统堆栈，这使得它们之间的切换只需很短的时间，于是在限时处理中有明显优势。

DSP/BIOS 内核提供一个特殊的软件中断线程（PRD）执行周期性函数。此线程基于时钟源运行，能实现周期性操作。

同步任务线程（TSK）是典型的应用程序执行线程。任务是唯一具有挂起特性的线程，这让它们在操作中有较高的灵活性。正因为这种灵活性，任务可以在不同的应用程序中使用。每个任务有私有堆栈，所以它们在上下文切换上需要花费的时间比软件中断长。因此，建议高限时要求的操作使用软件中断执行，低限时要求的操作使用任务线程。

第四种是后台空转线程。这个线程永远都在运行，在没有其他线程运行的情况下此线程不停地循环运行。

#### ➤ 标识互不相关的执行路径

使用 DSP/BIOS 线程为应用程序建立体系架构时，需要划分出来互不相关的执行路径。数据流图、系统块图和状态机都提供了不相关路径和相关路径的视图。典型的应用程序是使用不

同类型的线程构建的。用线程映射执行路径时，选择合适的线程类型以满足运行时的需求。线程类型的选择和许多因素有关，如优先级、延迟、负荷以及与其他线程间的依赖关系等。

### ➤ 标识实时截止时限和限时操作

实时系统对操作的完成有实时截止时间，应用程序中的限时操作也会有实时截止时限，因此它们的优先级必须以比非限时操作的优先级高。你必须决定系统中所有的实时截止时限，为每个操作设置相应的优先级。将操作映射到合适的 DSP/BIOS 执行线程时，这些信息是必需的。大部分限时线程会以高优先级的 SWI 或 TSK 线程执行。

除了优先级，线程延迟也可能影响对实时截止时限的满足。此处的延迟是指启动线程时上下文切换时间，需度量触发后启动线程所花费的时间。在 DSP/BIOS 内核中，硬件中断的延迟最短，即能最快地完成上下文切换。其次是软件中断。这是因为硬件中断和软件中断使用与应用程序相同的堆栈。每个 DSP/BIOS 同步任务有一个私有堆栈。这导致任务所需的上下文切换时间比软件中断长。

关于 DSP/BIOS 线程延迟和其他对象的详细说明和分析，请参考 Benchmarking DSP/BIOS on the C6000 (SPRA662) 和 Benchmarking DSP/BIOS on the TMS320C54x (SPRA 663)。

### ➤ 指定线程触发和同步

每个独立的执行线程都需要启动机制。硬件设备触发 HWI 管理的中断服务程序。软件触发 DSP/BIOS 的 SWI 和 TSK 线程。

每类 DSP/BIOS 执行线程使用唯一的启动和同步方法。DSP/BIOS 软件中断使用和硬件中断类似的模式运行。一旦触发，它们运行直到完成并停止。软件中断线程运行的优先级比硬件中断低，比同步任务高。软件中断根据 SWI 邮箱的内容来启动运行。应用程序可以使用这个接口来执行软件中断上的条件触发。软件中断线程可以被再激活，即在外部的控制下启动。通常，在事件触发软件中断前，软件中断函数所需的输入都必须是可用的。

同步任务 (TSK) 线程和中断有些不同。任务不可直接再触发。能在任务函数里使用 while 循环实现再触发。任务线程是唯一具有挂起属性的，这使得它们可以影响自己的运行。等待资源（如从外设获取数据或者访问共享内存）时，任务需要挂起，而不是轮询。挂起也可以通过睡眠一段时间或明确地放弃处理器的方式来中止任务。任务根据信号来同步执行。这使得任务线程的使用更灵活。使用信号可以根据事件或其他线程来同步任务的执行，也可以使得线程能公平地访问系统资源，如正常的执行路径、控制函数和非紧要的 I/O 都可以使用任务实现。

空闲循环以类似传统的主循环方式执行。循环中的每个函数按序执行，并且运行直到完成。你可以使用配置工具向列表里添加函数。在没有其他高优先级线程准备运行的情况下，空闲循环连续运行。传统的 ISR 和后台处理循环架构对于 DSP/BIOS 内核是有益的。首先要将传统循环函数移到空闲循环中。而将函数移到线程中可以提高性能。

所有 DSP/BIOS 线程都可以被高优先级的线程抢占。在设计程序时，需要为每个执行线程标识触发或启动函数，并判断是否希望在后台循环中执行应用程序的部分函数。

### ➤ 标识线程的有效期

DSP/BIOS 内核允许创建在应用程序的整个过程中都存在的线程。这些线程被称为**静态线程**。可以选择动态初始化 SWI 和 TSK 线程，即在应用程序需要时才存在；也可以删除它们来释放资源，这些线程被称为**动态线程**。需要标识应用程序中哪些线程是静态的，哪些线程会动态初始化。

使用配置工具创建静态线程。可以在应用程序中使用 DSP/BIOS API 创建/删除动态线程。静态线程的主要优点是减少内存需求。动态 DSP/BIOS 对象需要额外的代码来支持动态创建和删除，并且创建过程需要阻塞以等待完成内存的分配。

静态线程对实时分析工具的支持也比动态线程好。执行图不能为动态创建的线程直接建立可视化，也不能像静态线程那样显示细节信息。然而，在应用程序的设计中，动态线程具有更好的灵活性。

### ► 标识周期性或多速率操作

许多 DSP 应用程序周期性地处理数据。例如，音频数据采样和打包到 20ms 大小的缓冲区，则需要每 20ms 处理缓冲区。多速率系统意味着在不同速率执行多周期操作。DSP/BIOS 内核提供周期函数管理器 PRD，帮助你配置执行周期函数。

图 3.5 给出了周期性线程操作的说明。对于应用程序中的所有周期性操作，需要决定运行的操作周期和初始化后完成该操作的时间。所有的周期函数必须在 PRD 滴答的倍数时发生。因此，必须保证设定合适的 PRD 滴答周期。为了区分周期函数的优先次序，应该用 PRD 函数激发软件中断或同步任务来执行相应的操作。

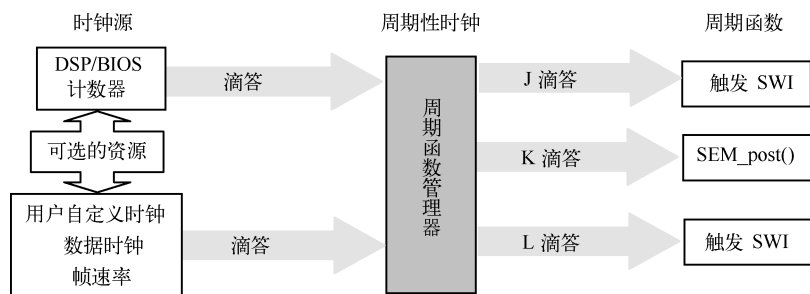


图 3.5 DSP/BIOS 周期性线程执行所有周期函数

有的应用程序执行周期操作来处理从外部来的数据，而不是用它们自己的时钟。例如，DMA 或 MCBSP 发送的帧数据，用帧的速率驱动时钟。如果应用程序需要执行周期操作这类数据，每次帧发送后，DMA 中断就会调用 PRD\_tick() 来驱动 PRD 时钟。

缺省设置是使用 DSP/BIOS 系统时钟来驱动 PRD 时钟，可以通过配置工具里的 CLK 管理器来修改设置。这使得周期函数以系统周期的倍数执行。

### 2) 标识数据通路和缓冲需求

在许多 DSP 应用程序中，输入到输出的数据流通常是连续的数据块或缓冲区。这种连续的数据流包括典型的音频、视频和语音应用。其他一些数据流可能是不连续的，如消息和通信通道。DSP/BIOS 数据管道（PIP）和数据流（SIO）都适合用来管理流数据。在应用程序中可以使用这些对象在 I/O 设备和应用程序或 DSP/BIOS 线程间传输缓冲数据。设计时，必须分离数据通路，每个数据通道可能包含多个线程。

流数据应用程序需要管理应用程序使用的数据缓冲区。典型的应用是 DMA 设备将数据在外设和系统内存间传输。如果数据输入时数据可用或输出数据时内存可用，DMA 传输完成会向应用程序发中断信号。应用程序中，并不直接对缓冲区进行管理。

PIP 模块提供了最小的体系架构，其代码高效、快速和灵活。可把 PIP 看成软件模式的 DMA。它完成 DSP/BIOS 线程间的数据传输，并提供事件驱动和轮询接口。和 DMA 设备几乎一样，当数据块传输完成时，它会触发一个中断。PIP 能运行回调函数通知目标线程数据

块已经可用。图 3.6 给出了 PIP 实现数据通路映射的典型方法。

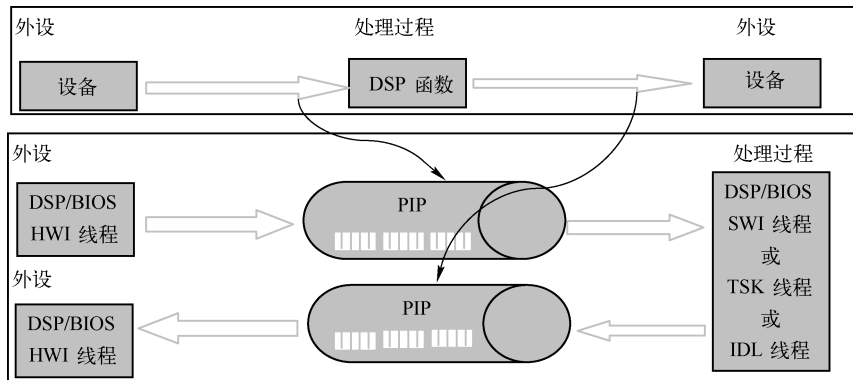


图 3.6 PIP 实现数据通路映射

SIO 模块提供高层次的机制实现同步任务线程和设备间的 I/O 流。SIO 提供了和底层设备不相关的简单通用接口。这个抽象层使应用程序设计者能简单地实现和其他设备的通信。

SIO 提供了创建和删除 I/O 通道、控制数据流（如启动、停止、空闲和刷新）的 API，以及在通道中传输数据的 API，如读取、写入、发布和收回。SIO 既支持静态创建，也支持动态创建和删除。图 3.7 说明 SIO 实现数据通路映射的典型方法。需要注意的是，SIO 链接的是设备驱动器和同步任务。这里用驱动器而不是设备，是因为设备驱动器分离了 SIO 和设备或线程。

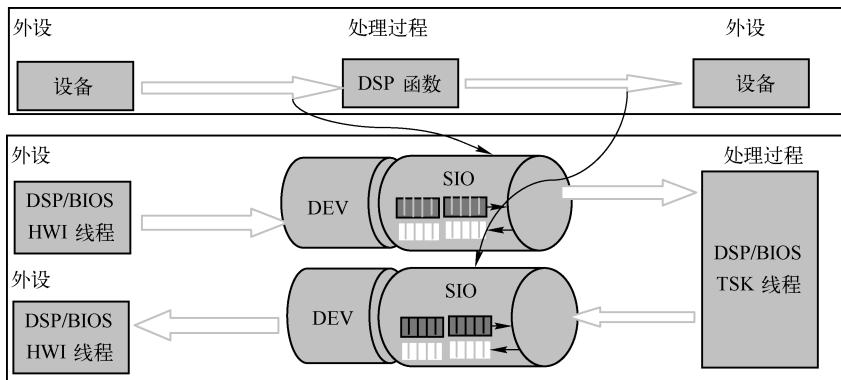


图 3.7 SIO 实现数据通路映射

PIP 和 SIO 都有灵活性，因为缓冲区的内容是用户自定义的。重要的是，PIP 和 SIO 是在线程间传递缓冲区的指针，它们并不拷贝缓冲的内容。这使得缓冲区的尺寸和内容独立于数据的传输，也就使传输时间固定，这对于实时系统是必要的。

要了解如何使用 PIP 和 SIO 构建多通道应用程序，请参考 Using PIP and SIO in Multi-channel Systems (SPRA689)。请参考 Writing Flexible Device Drivers for DSP/BIOS (SPRA700) 了解如何编写 PIP 和 SIO 接口所需的设备驱动。

标识线程内部的通信和消息通道：

典型的控制系统向任务发送消息命令它们操作，DPS/BIOS 内核提供邮箱在线程间发送消息。可以创建一个超级任务来接收和传输消息给系统中的其他处理任务。

邮箱与 PIP 和 SIO 不同，发给邮箱的消息需要拷贝到邮箱中。类似 PIP 和 SIO，邮箱中消

息的内容并没有严格规定。可以传输任何内容的消息，如命令、数据和内存缓冲区的指针。

DSP/BIOS 内核里的队列提供了线程间最简单的无拷贝的通信机制。DSP/BIOS 队列是双向链表，线程可以在队列中插入和删除元素。可以把队列看作软件实现的先进先出机制。队列并不是线程间的通信机制，但是，队列却是最常被使用的。实际上，SIO 在完成队列间通信和相同任务线程上的设备驱动时，使用了两个队列。

### 3) 标识 DSP/BIOS 对象

一旦完成了多线程设计，需要标识在应用程序中会使用的 DSP/BIOS 对象。在设计时，需要为每个执行路径选择合适的线程类型。如果线程间有依赖关系，就要决定所需的交互和同步。

还需要标识数据通路及其特征，即缓冲区大小和形式。I/O 设备通信，可以选择数据管道（PIP）或数据流（SIO）。线程内通信，可以使用数据管道、数据流、数据队列（QUE）或邮箱（MBX）。

### 4) 使用配置工具预配置系统和 DSP/BIOS 对象

使用 CCS 的 DSP/BIOS 配置工具可以配置目标系统 DSP/BIOS 内核的全局系统参数，包括 DSP 设备、CPU 时钟速度、字节顺序（大端/小段模式）、高速缓冲设置等。

- 使用配置工具选择和配置应用程序所需的 DSP/BIOS 运行时支持对象。使用这些对象，可以构造应用编程框架，来表示执行线程、I/O 及它们的接口，从而开发和验证应用程序的逻辑。
- 为 DSP/BIOS 线程指定线程优先级和激活线程时会调用的函数。还要为软件中断（SWI）指定传给激活线程函数所需的两个参数，以及初始 SWI 邮箱值。为同步任务（TSK）指定激活线程函数所需的参数，及任务堆栈的大小和占用的内存区。为在后台空闲循环运行的空闲函数指定要调用的函数。
- 使用配置工具生成系统内存视图。在不用 DSP/BIOS 的系统中，要设定链接命令行文件 CMD。而使用 DSP/BIOS 时，用配置工具指定 DSP/BIOS 内核要占用的内存区、系统堆栈的大小及占用的内存区。还可以选择性地设定 C 编译器所需的内存段。
- 使用配置工具创建中断向量表。在不用 DSP/BIOS 的系统中，通常要用汇编在单独的文件中指定这个向量表。
- 使用配置工具编程实现片上定时器，作为 DSP/BIOS 系统时钟。这个系统时钟是实时分析和其他 DSP/BIOS 时间函数所必需的。使用可视化界面来指定每次定时中断间隔的微秒数，配置工具可以生成所需要的寄存器设置。
- 使用配置工具，能通过片级支持库对外设进行编程，配置直接内存访问（DMA），多通道缓冲串行端口（McBSP）和其他外设。

图 3.8 总结了使用配置工具能实现的 DSP 设置，以及可以预配置的静态 DSP/BIOS 对象。

### 5) 在应用程序中调用 DSP/BIOS 应用程序接口

完成静态配置后，就可以在应用程序中调用 DSP/BIOS 的 API，访问和操作 DSP/BIOS 对象。

DSP/BIOS 内核使用应用程序的 main() 函数完成应用程序的初始化。可以把 main() 函数看成一个创建阶段，在这个函数里分配内存缓冲区，并完成其他的初始化工作，保证应用程序实际开始运行前能设置好所需的资源并使其可用。

main() 函数调用完成后，使能全局中断，DSP/BIOS 内核开始运行。



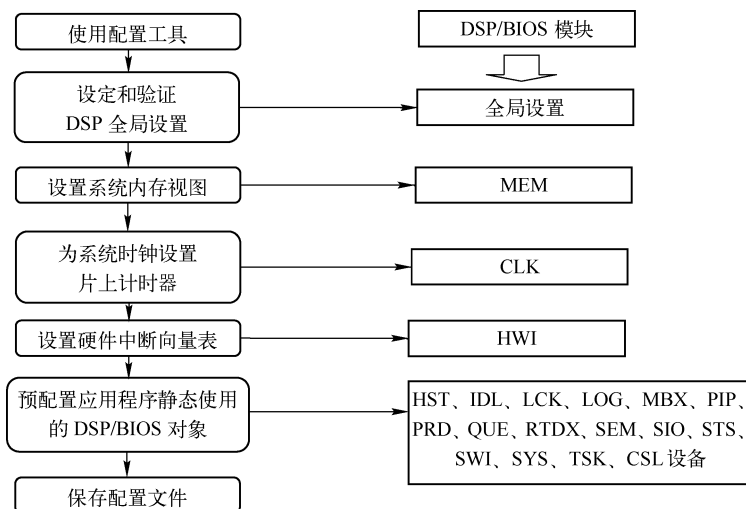


图 3.8 使用 DSP/BIOS 配置工具

### 3.1.3.2 音频应用示例

下面描述了一个使用 DSP/BIOS 开发的音频应用程序示例。它说明了如何使用 HWI、SWI 和 PRD 线程，也说明了如何使用数据管道（PIP 对象）管理应用程序运行期间的数据流。

该示例的第一步是分析功能需求，见图 3.9。

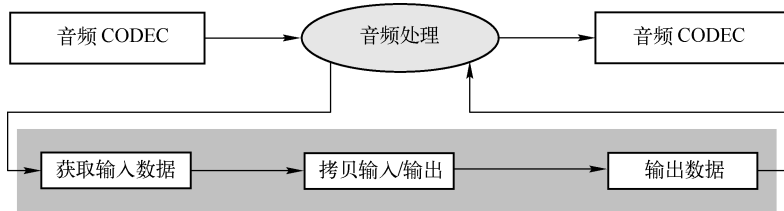


图 3.9 音频处理功能需求图

应用程序需要从 CODEC 获取音频数据，执行一些处理操作，再将数据输出给 CODEC。作为简化，下面的示例只是将音频输入拷贝给输出，见图 3.10。

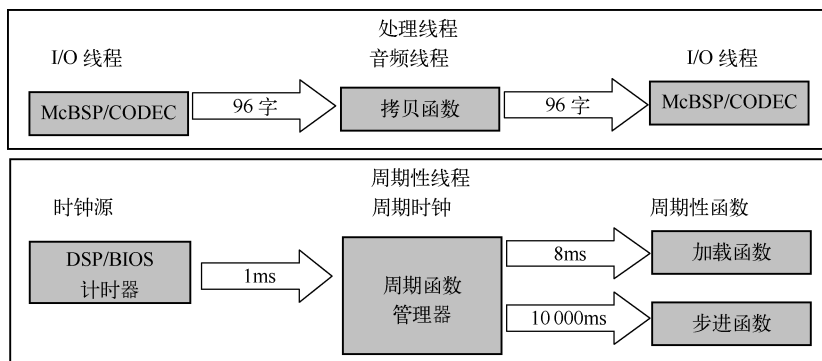


图 3.10 音频示例模块图



除了音频处理外，你需要周期性地加载 CPU 来浏览音频处理的影响。加载功能可以增加和减少，以调整应用程序的负荷。见图 3.10 中的周期性线程。

现在，已经有了划分执行线程的模块图，可以开始分配 DSP/BIOS 线程了。由于这个音频处理过程非常简单，只需要使用一个线程来完成。而且采用了相同输入/输出速率的简化设计，因此，使用同一个 ISR 来处理输入和输出。使用 HWI 来管理中断。SWI 或 TSK 线程都能用来完成音频处理，但是，在这个示例中使用的是 SWI，以说明如何使用 SWI 邮箱来同步多触发源。只有一个使用 SIO 模块的 TSK 线程，因此不需要同步。

当只存在一个满输入数据的缓冲区时，才需要 SWI 来处理音频，并且只有一个可用的空缓冲区来存储处理过的数据。使用数据管道来管理缓冲区流，说明了回调方法如何触发 SWI 音频处理。图 3.11 示范如何将一些 DSP/BIOS 对象映射到音频处理线程上。因此，在这个简单例子中，有一个 HWI 线程处理 CODEC 的中断，一个 SWI 线程用来负责音频处理。中断处理是一个执行线程，音频处理是另一个执行线程。连接这些执行路径或线程的是管理缓冲区流的数据管道。

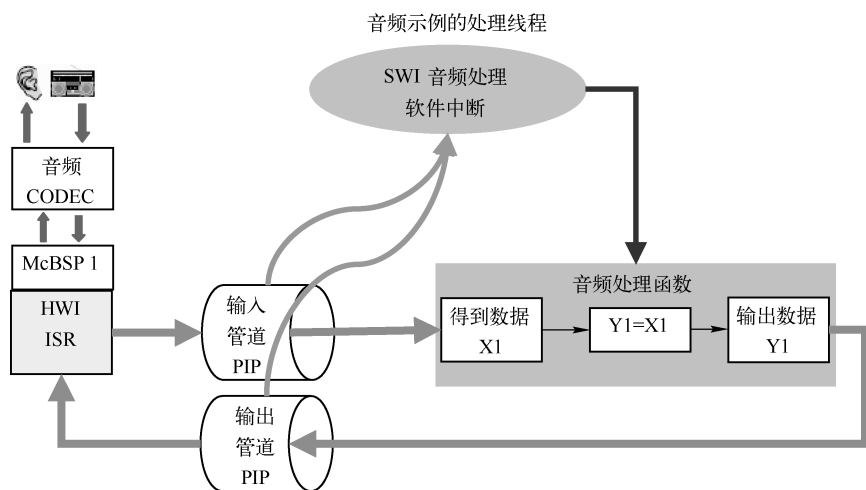


图 3.11 音频示例映射为 DSP/BIOS 线程

另外，与音频处理不相关，示例中还有加载 CPU 的周期性函数。系统时钟每 1ms 驱动 PRD 时钟一次。在这个示例中，PRD\_swi 线程中直接执行 PRD 函数。每 8 毫秒加载函数运行一次；每 10 000ms (10s) 运行一次步进函数。步进函数用于修改加载函数，以增加或减少加载周期。

硬件平台以 TMS320C6201 EVM 为例，同时参考 TMS320C6211 DSK、TMS320C6711 DSK 和 TMS320C5402 DSK。如图 3.12 所示，多通道缓冲串行端口 McBSP1 连接到音频 CODEC。使用 ISR (DSS\_isr()) 和 McBSP 交互，读写音频数据，使用 DSP/BIOS HWI 模块来管理硬件中断。CODEC ISR 用输入数据填满空缓冲区块，输出满缓冲区块直到缓冲区块空。

如图 3.12 所示，由 PIP 模块管理 DSP/BIOS 数据管道，在 ISR 和应用程序之间传输数据。一个数据管道从 ISR 传输数据给应用程序 (DSS\_rxPipe)；另一个传输数据给 ISR 用于输出 (DSS\_txPipe)。

DSP/BIOS 软件中断线程 (audioSWI) 使用 audio() 函数来完成音频处理。仅当满数据块和空数据块都准备好时，才激活音频处理线程。使用 audioSWI 的邮箱来同步这些事件。

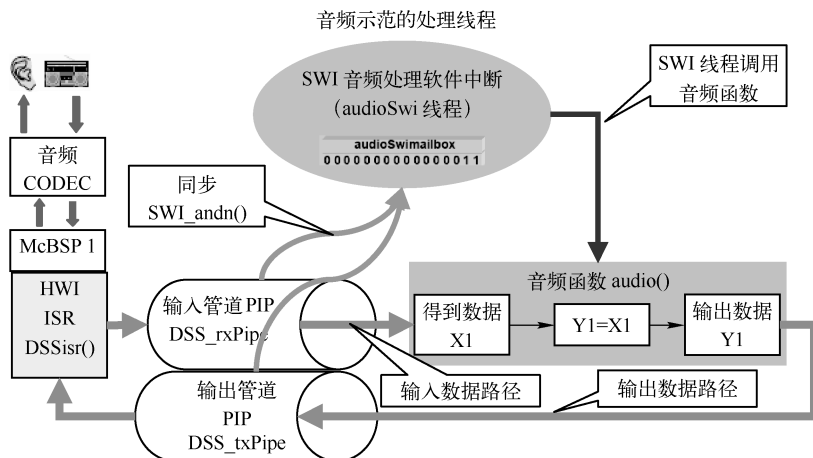


图 3.12 使用 DSP/BIOS 内核进行音频处理操作

SWI 邮箱的初始值设为 3，其头 2 位设为 1。当这两位都变为 0（邮箱的值为 0），激活 SWI 线程完成音频处理。要使用数据管道通知函数分别将这些位清零。

两个数据管道都调用 SWI\_andn() 产生软件中断，以设置 SWI 邮箱里的位来同步音频处理。当 ISR 已经填满了一个数据块，输入数据管道就会发信号 audioSWI，调用 SWI\_andn(2) 来清空 SWI 邮箱里的第 1 位，使数据可用于处理。类似地，当空数据块可用时，输出数据管道会发信号 audioSWI，调用 SWI\_andn(1) 来清空 SWI 邮箱里的第 0 位。

图 3.13 所示为周期性线程的配置。注意，使用系统时钟来驱动周期时钟，周期是 1ms。PRD 管理器会每 8ms 调用 load() 一次，每 10s 调用一次 step() 函数。

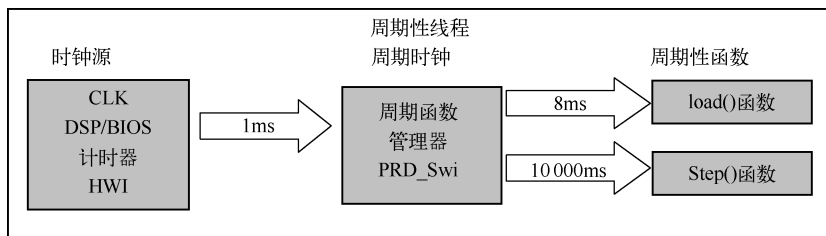


图 3.13 使用 DSP/BIOS 内核加载周期性处理操作

这个示例中，使用的 DSP/BIOS 对象都是静态的。文件 audio.cdb 保存了配置信息。在应用程序中，可以观察用来访问数据管道中缓冲区、ISR 和 audio() 中的 API 调用。

### 总结

将应用程序分成多个独立线程后，就需要赋予合适的 DSP/BIOS 线程类型。DSP/BIOS 内核提供了四种基本线程类型，可以根据执行的需求选择。

然后，选择合适的数据流对象来传递应用程序所需的数据。DSP/BIOS 内核提供了设备无关的 I/O 对象处理连续的块数据传输。为线程内部通信和其他数据传递，DSP/BIOS 内核提供了邮箱和数据队列。

DSP/BIOS 内核提供了系统配置工具来指定和配置系统组件，如系统内存视图和硬件中断向量表。也可以使用这个工具静态声明应用程序中会用到的 DSP/BIOS 对象（线程和 I/O）。

### 3.1.4 DSP/BIOS 的编程和调试

DSP/BIOS 提供了丰富的工具,帮助 DSP 应用程序的开发者设计和开发实时嵌入式应用程序。本节讨论了容易发生错误的地方及解决策略,以及如何使用 DSP/BIOS 编程。下面从 8 个方面讨论了编程相关问题,这些问题并不和特别的硬件有关。第 5 章将介绍 CCS 调试工具。

#### 1. 启动

CCS 启动后会运行 GEL 文件初始化目标主板。由于内存配置的改变,以及不同 DSP 和主板有不同的初始化需求,GEL 文件里的 StartUP 程序通常是以注释的方式、随时可修改的。首先要做事情是审视这个文件,根据自己的主板和 CPU 定制相应的初始化设定。可以右键单击 CCS 桌面图标获得 GEL 文件的地址和名称。

可以使用 GEL 下拉菜单上可用的复位(reset)选项重启硬件,这不仅可以重启 CPU,也可以重启主板上的外设。而 Debug 下拉菜单上的 DSP 重启功能只能重启 DSP,并不能重启主板外设。另外,可以创建多个 CCS 的快捷键来设定启动设置,每个启动设置在命令行中使用一个不同的 GEL 初始化文件。

注意,在开发过程中,C54x EVM 主板偶尔会进入一个状态,导致重启操作无法工作。CCS 在访问主板或 JTAG 硬件、清除断点的时候会报错。当这种情况发生时,退出 CCS,按 EVM 的重启键,重新启动 CCS。如果这样还不能解决这个问题,就需要使用 XDS-510 或/和 EVM 主板上的电源硬启动。

C6x 目标机的默认 GEL 文件是和 CCS 的版本相关联的,文件名可能是 init.gel 或 c6x.gel。无论怎样,都是配置 C6x EVM 主板的。保存这个文件,并退出 CCS。当要重启 CCS 时,就会进行初始化,初始化访问主板上外存的接口。

如果在除了 EVM6x 而外的其他主板上工作,就要将这个文件的内容作为模板,按照不同主板的需求修改该文件。更多的信息参见特定主板的文档和 TMS320C6000 Peripherals Reference Guide。

#### 2. 堆栈

在任何系统中,关于堆栈有两个基本的概念:溢出和泄漏。DSP/BIOS 提供一些工具监控堆栈的使用,能标记溢出或泄漏的发生。

在 DSP/BIOS 里,有三种基本的执行线程类型:硬件中断 ISR (HWI)、软件中断 (SWI) 和任务 (TSK)。系统的堆栈有两类:任务堆栈和中断 (ISR) 堆栈。每个任务有自己的私有堆栈,而 SWI 和 HWI 则没有。ISR 堆栈也是有的文档里提到的系统堆栈,通常,它是一个所有中断都使用的公共堆栈。在硬件相关的部分会讨论它的局限性和上下文切换中两个堆栈的角色与任务。

##### 1) 查看堆栈

可以在内存视图 (Memory View) 窗口中查看堆栈。根据查看的堆栈类型,在内存里定位堆栈。

使用指针 (.stack) 在自己的内存段放置 ISR 堆栈。也可以在内存属性 (Memory Properties) 窗口的地址栏里输入一些标记/符号访问 ISR 堆栈。

GBL_stackend	:堆栈缓冲区的开始指针
HWI_STKBOTTOM	:可用堆栈的开始(底部)指针(堆栈的头指针)
HWI_STKTOP = GBL_stackbeg	:堆栈的终止(顶)指针

标号 GBL\_stackend 可能和 HWI\_STKBOTTOM 有些许不一样。GBL\_stackend 定义了堆栈缓冲区的开始位置, HWI\_STKBOTTOM 是堆栈指针可以指向的最大值。HWI\_STKTOP 和 GBL\_stackbeg 都指向同一个值, 堆栈缓冲区的结束位置。在内核/对象视图中, ISR 堆栈的信息也是可访问的。

任务堆栈信息, 如开始和结束地址、最大使用量, 也都可以内核/对象视图中查看。在内存属性窗口的地址行里输入这些地址, 就能查看这个堆栈。另一个查看静态分配的任务私有堆栈的方法是在内存属性窗口的地址栏里输入 <taskname> \$stack。

也可以使用 DSP/BIOS 的 API (TSK\_stat) 获取这些信息。

## 2) TSK\_checkstacks()

DSP/BIOS 的 API 函数 TSK\_checkstacks() 可以检测任务堆栈是否溢出和泄漏。具体的调用语法如下。

```
Void TSK_checkstacks (TSK_Handle oldtask, TSK_Handle newtask);
```

在上下文切换时, 调用这个 API, oldtask 是被切换的任务, newtask 是要切换到的任务。

任务创建时, DSP/BIOS 会初始化每个任务的堆栈, 在每个堆栈的所有位置都写入一个已知的值。对所有的任务堆栈, 这个值都是相同的, 在 DSP/BIOS 里使用 TRG\_STACKSTAMP 引用它。处理器家族之间, 这个值会不同 (具体参见硬件相关部分)。

TSK\_checkstacks 使用任务句柄获取两个堆栈的地址和大小。然后, 检查末尾是否是 TRG\_STACKSTAMP。如果末尾不是 TRG\_STACKSTAMP, 表明它已经被重写。要清楚哪一个堆栈测试失败: 如果 oldtask 堆栈测试失败, 说明 oldtask 堆栈溢出, 则问题发生在 oldtask。如果 newtask 任务测试失败, newtask 堆栈因为一个不合适的写操作发生了泄漏问题。由于 newtask 并没有运行, 因此是一个其他任务出的错误。在另一个事件中, 会调用 SYS\_abort 产生错误信息。

**注意:** 虽然堆栈都被 TRG\_STACKSTAMP 填满了, 测试的位置只是堆栈的最后一位。

应用程序也可以直接调用 TSK\_checkstacks, 但是在上下文切换时这个 API 更有用。每个任务上下文切换时要检查每个切换的堆栈的完整性。DSP/BIOS 配置工具也可以进行这个检查。在任务管理器中, 设置调用切换函数 (Call switch function) 的属性为 True, 并设置切换函数 (Switch function) 为 \_TSK\_checkstacks 即可。

如果堆栈溢出, 就应该手动检查堆栈, 判断溢出的范围或程度以及下一个动作。溢出可能表示开发者对这个任务预估的堆栈大小不正确, 或者任务代码有缺陷/错误。实验堆栈大小, 使它足够大, 观察错误是否再发生, 尽量判断随着堆栈增减时实际所需要的堆栈空间大小。一旦决定了需要的实际空间大小, 就能核查错误预估的原因。另一个方法是一开始多分配堆栈, 完全运行整个系统, 根据最后堆栈的实际使用量来分配。

根据配置中静态定义的 DSP/BIOS 对象, DSP/BIOS 配置工具自动预估了系统 ISR 堆栈所需的最小尺寸。可以通过编辑内存段管理器 (Memory Section Manager) 属性窗口里的堆栈尺寸 (Stack Size) 来修改这个最小值, 存储堆栈的内存地址也可以在这个窗口里修改。

堆栈管理属性窗口指定了缺省堆栈尺寸和存储段。但是，每个任务可以有自己的堆栈尺寸和存储段，也可以在这个窗口里修改每个任务的堆栈尺寸和存储段。

增加堆栈的空间可能是解决方案，也可能不是。由于 DSP 的系统里，对内存的约束是很严格的，在任意增加堆栈大小前，必须首先明白堆栈溢出的原因。有的情况是 TSK\_checkstacks 遗漏了错误或者错误报错，因此，手动检查和实验总是必需的。

堆栈泄漏是一个非常难以解决的问题。需要认识的重要事情是当前任务不会造成这个错误。首先，需要手工检查，了解是否是一个常见的、可辨识的泄漏模式。许多时候，这能快速地区别问题源。下一步是检查每个用来引用的指针取值。使用日志工具记录指针使用前的取值，确保初始化正确。在 CCS 中有一些工具，如仿真分析工具，提供了内存访问的硬件断点。在内存部分会讨论这个工具。

另一个最常引起内存泄漏的原因是写数组越界。对一个数组进行写操作时，必须检查下标，以保证在该数组声明的内存区域里写数据。对于最终的应用程序而言，可能需要经常检查边界，可以使用断言（ASSERT）在调试阶段进行边界检查，调试完成时并不编译到最终的应用程序中。

**TSK\_checkstacks 的局限：**TSK\_checkstacks 有一些潜在的问题。它只能检查单独的任务，不能检查 ISR 任务。

如果堆栈的最后一个字被覆盖就会报告溢出错误。技术上而言，这可能不是溢出，因为那是堆栈上最后一个可以写的位置。此时，正好使用了堆栈声明的所有空间，尽管并不建议这样严格声明堆栈空间。为堆栈增加额外的存储空间会消除这个问题，并且能给你需要的空闲空间。建议额外多分配 10% 的堆栈存储空间来处理可能发生的不可预料的情况。

由于堆栈会为了调整基准位的目的预留一些空洞，因此溢出有可能发生但是没有被检测到。堆栈中，可能会分配位置但是并不使用。在这些情况中，这些位置就会依然是 TRG\_STACKSTAMP 的值。如果 TRG\_STACKSTAMP 依然在原处，堆栈可能会继续溢出，但是并不能被检测到。

### 3) TSK\_stat()

TSK\_stat() 是一个用来查询任务的状态数据（包括任务堆栈的状态数据）的 DSP/BIOS API，其调用语法是：

```
void TSK_stat (TSK_Handle task, TSK_Stat * buf);
```

使用缓冲区保存任务当前状态的数据结构，其中包含的与堆栈相关的数据项是：当前堆栈指针、堆栈使用的介质链接单元（MAU）最大数、堆栈基地址、堆栈的内存段、声明的堆栈大小。如果使用的数目等于堆栈的大小，则堆栈溢出。任务能检查自己或其他任务的状态，如果堆栈溢出或接近溢出时，任务可以采取行动。返回的缓冲区中实际的数据结构如下。

```
struct TSK_Stat {
    TSK_Attrs      attrs;      /* 任务属性 */
    TSK_Mode       mode;      /* 任务执行模式枚举结构 */
    Ptr            sp;         /* 任务堆栈指针 */
    Uns            used;       /* 任务堆栈使用状态 */
};
struct TSK_Attrs {
```



```

    Int    priority;    /* 运行优先级 */
    Ptr    stack;       /* 预分配的堆栈地址 */
    Uns    stacksize;   /* MAU 里的堆栈尺寸 */
    Int    stackseg;    /* 堆栈分配的内存段 */
    Ptr    environ;     /* 全局环境数据结构 */
    String  name;       /* 可打印的名称 */
    Bool    exitflag;   /* 程序中止需要该任务中止的标记 */
    TSK_DBG_Mbxdbg;     /* 调试枚举结构 */
};

```

关于这个 API 的数据结构的具体信息，请参见 DSP/BIOS 用户手册。

#### 4) HWI 监控属性

系统里的每个 HWI 都有一个相关联的属性，名为 monitor。可以使用这个属性监控数据的取值、系统寄存器和堆栈指针。STS 对象和被监控的 HWI 相关联。这允许用户打开 CCS 里的统计视图，观察最大值、最小值，以及应用程序运行时被监控数据的取值。可以在配置工具里 HWI 窗口的属性窗口里访问这些特性。**注意：这个工具只在不使用任务的系统里使用。**

知道了 HWI 的监控属性何时可用，可以在 HWI ISR 里插入代码脚本，即可进行监控。每个被监控的 HWI 每次中断须运行 20 ~ 30 个指令完成监控。不推荐在正式代码中开启该指令，因为 HWI 处理是系统中对时间最敏感的部分。

#### 5) 内核/对象视图

内核/对象视图可以让用户检查系统中所有 DSP/BIOS 静态或动态对象的状态。内核/对象视图里，能浏览 DSP/BIOS 内核 (KNL)、任务 (TSK)、邮箱 (MBX)、信号 (SEM)、内存段 (MEM) 和软件中断 (SWI) 的状态信息。

对于 KNL 对象，可以浏览下列信息：

- ISR 堆栈的开始位置、结束位置、大小和被使用的最大数目；
- 当前被阻塞的任务；
- 内核模式；
- 进程 ID；
- 当前时钟。

对于 TSK 对象，可以浏览下列信息：

- 任务名称和句柄；
- 任务堆栈开始位置；
- 任务堆栈结束位置；
- 任务堆栈中被使用的最大空间数；
- 任务状态；
- 任务优先级。

对于 MBX 对象，可以浏览下列信息：

- 邮箱的名称和句柄；
- 每个邮箱的当前消息数和最大消息数；
- 消息的大小；
- 等待读邮箱的任务；



- 等待写邮箱的任务；
- 邮箱使用的内存段。

对于 SEM 对象，可以浏览下列信息：

- 信号的名称和句柄；
- 数目；
- 等待的任务。

对于 MEM 对象，可以浏览下列信息：

- 内存区（堆）的名称；
- 堆里的最大连续空间；
- 空闲空间；
- 开始地址；
- 结束地址；
- 被使用的数目；
- 内存段。

对于 SWI 对象，可以浏览下列信息：

- SWI 的名称和句柄；
- 状态；
- 优先级；
- 关联的邮箱；
- 关联的函数和参数。

特别的数据是堆栈的信息。对于系统堆栈和任务堆栈，如果检测到溢出，被使用的最大空间数（Max used 或 peak used）所在的浏览框的颜色就会变红，里面的文字颜色会变黄。但是，需要注意的是，和 TSK\_checkstacks 类似，堆栈中存在用来校正基准位的空块可能会导致溢出检测错误。

#### 6) 上下文切换时堆栈的用法

如果中断（HWI 或 SWI）抢占了任务，就要在 ISR 堆栈上保存任务的上下文，并且在 ISR 处理该中断。这里，任务的上下文是指任务被中断时 CPU 寄存器里的内容。

图 3.14 中，任务在自己的堆栈上运行，直到在点 1 处被中断。在点 2 处，堆栈指针被修改为 ISR 堆栈的底部。被中断的任务的上下文被存放在 ISR 堆栈上，堆栈指针指向点 3 处，即 ISR 处理程序实际的开始点。内存中，堆栈继续向下分配（分配到低地址位）。

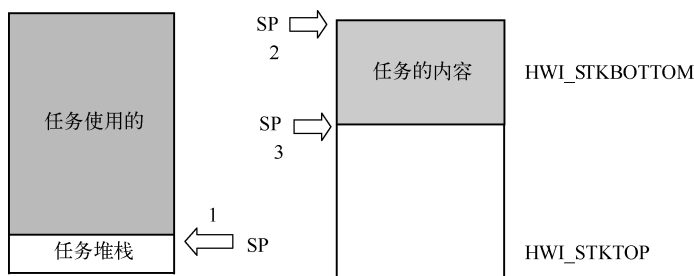


图 3.14 任务被中断抢占

当中断处理完成时，如果之前正在运行的任务需要被恢复，就需要重新存储其上下文，之后，该任务会在自己的堆栈上继续运行。

图 3.15 中，所有的中断处理都已经在点 1 处完成。从 ISR 堆栈中取出被中断的任务的上下文，堆栈指针移动到点 2 处。然后重新存储任务的上下文，将堆栈指针恢复到中断时的取值，即点 3 处。

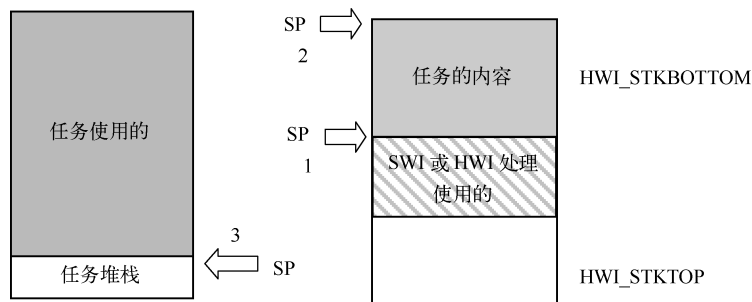


图 3.15 中断后恢复任务的过程

但是，如果被抢占的任务被阻塞，另一个其他的任务准备运行，就需要将保存的上下文从 ISR 堆栈转存到被抢占的任务堆栈里，从新任务堆栈里获取其上下文，在任务堆栈上开始运行新任务。

图 3.16 中，在点 1 处，所有的中断处理都已经完成。内核认为任务 2 应该运行，所以在点 2 处，把被中断的任务的上下文从 ISR 堆栈转存到第一个任务的堆栈中。然后，在点 3 处，从任务 2 的堆栈中获取并存储其上下文，将堆栈指针移到任务 2 被阻塞时的位置，即点 4 处。

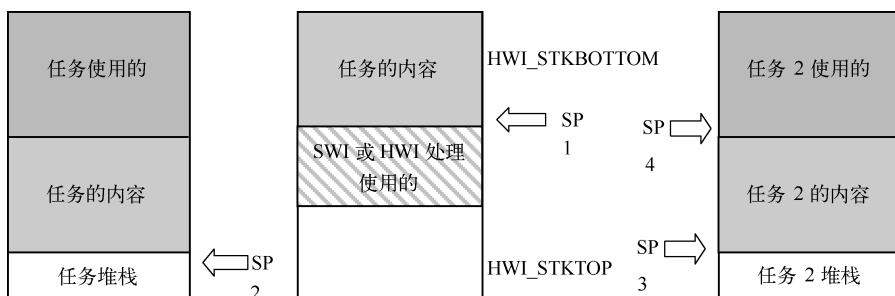


图 3.16 中断处理后任务被阻塞

如果系统不使用任何 TSK 对象，那么就没有任务堆栈，所有中断处理都在 ISR 堆栈上发生。

#### 7) 关于 C54x

在 C54x 上，最小寻址单元（MAU）是 16 位的。按照 C 程序的习惯，堆栈基准是 2 个 MAU。当 C 和汇编之间接口链接时，必须由用户的代码维护堆栈基准。堆栈的指针寄存器是 SP 寄存器。

与前面讨论的一样，每个堆栈的顶部都是由一个特殊值标记的，即 TRG\_STACKSTAMP，TSK\_checkstacks 使用这个值判断溢出和泄漏。在 C54x 上，这个值是 0xBEEF，保存在 ISR 堆栈中。

C54x 上已知的堆栈和内存泄漏的潜在来源是汇编调用 DSP/BIOS 的 API 和 C 程序前不正确的 CPL 和 DP。

#### 8) 关于 C6x

在 C6x 上，最小寻址单元（MAU）是 8 位。C 程序中，堆栈基准是 8 个 MAU。C 和汇编之间接口链接时，必须由用户代码维护基准位。堆栈指针寄存器是 B15。

如前所述, TRG\_STACKSTAMP 是标记堆栈顶部的特殊值, TSK\_checkstacks 使用它检测溢出和泄漏。在 C6x 上, 只有任务堆栈使用 TRG\_STACKSTAMP, ISR 堆栈用另外的值标记。这些常量的取值分别是:

TRG_STACKSTAMP:	0xBEBEBEBE;
ISR 堆栈顶:	0x00C0FFEE。

### 3. 硬件中断

中断是实时系统的关键。它们是描述现实世界中在 DSP 上发生的事件, 如按下按键、丢弃信号、获取信号、新数据到达。因此, 中断处理是嵌入式系统中一个最复杂、最关键的问题。在 DSP/BIOS 系统中, 硬件中断 (HWI) 管理器负责处理中断。简而言之, 不能在 HWI 中调用任何创建或删除系统对象、可能死锁系统资源 (如信号、邮箱等) 的 API。DSP/BIOS 用户手册的附录 A 包含了 DSP/BIOS API 的完整列表以及它们的上下文制约, 包括 HWI 中可以调用哪些 API 和不能调用哪些 API。编程时应参考此列表, 确保 ISR 只执行合法的操作。

如果应用程序中需要使用这些 API, 那么应该在 SWI 或任务中调用。

如果 ISR 不调用 DSP/BIOS, 就不会有其他特殊需求, 就不会发生上下文切换的行为。

#### 1) 使用 DSP/BIOS 的 ISR

如果 ISR 要调用 DSP/BIOS, 就必须满足一些需求。C 或汇编编写的 ISR 中, 必须调用两个宏 HWI\_enter 和 HWI\_exit。在任何 DSP/BIOS 调用前, 必须先调用 HWI\_enter, HWI\_exit 必须是 ISR 最后执行的语句。应在 ISR 末尾, 使用 HWI\_exit 代替 return 语句。这两个宏一起为 ISR 实现下列功能:

- 保存和存储当前 CPU 寄存器的内容 (为 C 调用特定的 DSP/BIOS API 保存内容);
- 禁止和允许指定的中断 (控制中断的嵌套);
- 如果中断嵌套发生, 只有在第一个外层的 ISR 完成后, 调用 DSP/BIOS 调度程序。

可以使用 HWI 调度器代替 HWI\_enter 和 HWI\_exit。当你要用 C 语言编写 ISR 时, HWI 调度器是非常有用的。

**警告:** 绝对不要在 DSP/BIOS 程序中, 使用编译器的关键词 “中断” (interrupt)。在 DSP/BIOS 应用程序中使用这个关键词会产生不可预知的行为。相反, 应使用 HWI\_enter 和 HWI\_exit 宏或 HWI 调度器。

触发 HWI 时, 所有中断被 DSP 禁止, 然后处理程序会跳转到中断向量表中的 ISR。根据调用中指定的中断掩码, HWI\_enter 只禁止相应的中断。最后, HWI\_enter 重新启动所有没有屏蔽掩码标记的中断。

调用 HWI\_exit 时, 它也禁止所有中断。然后, 使用指定的中断掩码, 重新启动调用 HWI\_enter 时禁止的中断。HWI\_exit 只会重新启动指定的中断, 并且只是之前 HWI\_enter 中禁止的中断。最后, 再重新启动所有中断。

#### 2) 中断和流水线

汇编程序比 C 语言的程序有更多细节的处理。为流水线的 DSP 写汇编代码是一组自己特有的指令。这里不是讨论处理器的流水线问题, 而是讨论汇编语言的编程者如何使用中断和流水线的交互实现, 以及如何处理这些交互。下面的例子针对 C6x 和 C54x 的用户。

**例 1** C6x 中的例子:

```

1 LDW  *A5 ++ ,A2
2 LDW  *A6 ++ ,A2
3 SUB  B4,B1,B2
4 NOT  B2,B4
5 NOP
6 ADD  A2,A3,A4
7 MV   A2,A3

```

### 例 1A

这个代码示例可以是流水线代码优化后的结果。虽然第 1 行和第 2 行的装载指令装载了相同的寄存器 A2，但是每条装载指令在 4 个延迟槽后才生效。对于第一条装载指令，在第 5 行后，值才在 A2 中。对于第二条装载指令，在第 6 行后，值才在 A2 中。因此，在第 6 行的加法指令使用的是第 1 个装载到 A2 的数值。第 6 行后，在 A2 装载了第二值，所以第 7 行是将第 2 个值移到 A3 中。

这代码通常运行得很好，直到发生中断。如果在第 2 行和第 6 行之间发生了中断，两条装载指令都会在第 6 行指令执行前完成。处理器中断时，流水线上，之前取到 E1 的指令都会完成，所以在通常的 4 个延迟槽之后，每条装载指令都会完成。这发生在 ISR 处理过程中。ISR 完成后，A2 已经被第 2 个装载指令写覆盖了，所以第 6 行将是 A3 与这个数值相加，产生错误。第 7 行正常执行，因为这一行本身就是在第二个装载的数值上操作的。但是，最后的结果是 A4 中的值是错的，它是由 A2 中的错误数值计算得来的。如果中断发生在第 1 行和第 2 行之间，处理过程会正常完成，因为 ISR 完成的时候 A2 中装载的是第一个数值。如果没有其他的中断，代码会正确完成。

C54x 中类似的例子是：

```

1 STLM A,AR2
2 STLM B,AR2
3 NOP
4 LD  *AR2,A
5 LD  *AR2,B

```

### 例 1B

完成第 1 行和第 2 行需要两个延迟槽，意味着如果中断发生在第 2 行和第 4 行之间，同样的问题会发生在 C6x 上。

这段代码是双重赋值的示例，即使用相同的寄存器保存两个不同的数值，其中的 1 个或 2 个数值在流水线上等待处理。在双重赋值中，定时和指令序列是严格的。任何中断都会影响定时和指令序列，也会因此产生错误的结果。这对于调试是非常困难的，因为按步跟踪代码可能是正确的结果。主代码和 ISR 可以单独完全正确地执行。这是实时交互中发生的错误。

如何解决这个问题呢？有两种方法。一种方法是使用单赋值重写这段代码。双重赋值使用同一个寄存器保存一个数值，另一个数值在流水线上等待，单赋值则是不要让寄存器的数值在流水线上等待。基本上，单赋值就是，所看到的的就是所得到的。下面是用单赋值重写示例 1A：

```
1 LDW * A5 ++ , A2
2 LDW * A6 ++ , A7
3 SUB B4, B1, B2
4 NOT B2, B4
5 NOP
6 ADD A2, A3, A4
7 MV A7, A3
```

### 例 1C

上面这段代码执行效果和第 1 段代码完全一样。只是，当中断发生在第 2 行和第 6 行之间时，之前的代码此处是“问题区域”。ISR 完成时，两条装载指令都已完成，A2 和 A7 被装载了数据，在第 6 行和第 7 行分别完成相应的操作。这些操作的执行不会产生错误，完全都是正确的。这个解决方案的不利处就是使用了一个额外的寄存器。在有些场景中，这可能不能实现节约资源的目的，因此需要进行权衡。

下面是对例 1B 用单赋值重写的代码：

```
1 STLM A, AR2
2 STLM B, AR3
3 NOP
4 LD * AR2, A
5 LD * AR3, B
```

### 例 1D

另一个解决方法是，在使用双重赋值时禁止中断。这是编译器和汇编优化器在处理软件流水线循环时使用的方法，即，在关键代码段前禁止中断，在其后开启中断。

#### 3) HWI\_disable、HWI\_enable 和 HWI\_restore

如前面的例子所示，在 C54x 和 C6x 上，可以巧妙地禁止中断。但是，对每个处理器家族而言，这个过程有些不同。因此，DSP/BIOS 提供了一个核心的机制来处理中断的开启和禁止。这个机制保证对所有的处理器都是起作用的，且使用方式一致。

HWI\_disable 用来禁止全部有屏蔽掩码的中断，方法及效果和上面的例子一样。它保证不被中断影响，让操作总能正确完成，并且程序员不需要编写针对特定处理器的代码。HWI\_disable 返回禁止中断前的全局中断开启标记的值（也包含一些其他的状态标记）。该数值会在后面 HWI\_restore 中使用。

HWI\_enable 用来开启所有中断的，并不考虑全局中断位或 PGIE 位的状态。如果中断允许/禁止是嵌套的，一定不要使用 HWI\_enable。中断允许位之前的状态不会被保存。

HWI\_restore 存储了调用 HWI\_disable 前全局中断位的状态。HWI\_restore 在调用前也要禁止中断，HWI\_enable 没有这个要求。HWI\_enable 没有任何参数，HWI\_restore 需要之前的 CSR 状态作为参数，该参数是调用 HWI\_disable 时获得的。如果调用是嵌套的，可以使用 HWI\_restore。

### 例 2

对上面的 C6x 和 C54x 例程，最后的推荐方案如下：

```

1  HWI_disable B0
2  LDW * A5 ++ , A2
3  LDW * A6 ++ , A2
4  SUB B4 , B1 , B2
5  NOT B2 , B4
6  NOP
7  ADD A2 , A3 , A4
8  MV A2 , A3
9  HWI_restore B0

```

### 例 2A (C6x)

```

1  HWI_disable * AR1AR1
2  STLM A , AR2
3  STLM B , AR2
4  NOP
5  LD * AR2 , A
6  LD * AR2 , B
7  HWI_restore * AR1AR1

```

### 例 2B (C54x)

注意，在这两个例程中，如果不需要嵌套时，可以用 HWI\_enable 替换 HWI\_restore。

#### 4) 关于 C54x

在 C54x 上，在 ISR 末尾使用 HWI\_exit 有一个额外的好处。如果用远程内存模型构建应用程序，HWI\_exit 会自动执行远程返回。在内存部分会讨论远程内存模型。

#### ➤ HWI\_enter 和 HWI\_exit 的前置条件

这两个宏的前置条件非常重要。HWI\_enter 的主要后置条件（这个宏设定的条件）就是 HWI\_exit 的前置条件。HWI\_exit 的前置条件是：

```

cpl = ovm = c16 = frct = cmpt = 0 (cpl 是编译模式位；ovm 是溢出模式位；
                                c16 是运算器 (ALU) 精确模式位；
                                frct 是碎片模式位；cmpt 是兼容性模式位；
                                在 ST1 寄存器中装载所有位)
dp = GBL_A_SYSPAGE (dp 是数据页指针，存放在 ST0)

```

关于这些前置条件的意义和细节参见 DSP/BIOS API 前置条件部分。调用 HWI\_enter 的唯一前置条件是必须禁止所有中断。跳转到 ISR 前，在响应当前中断时，DSP 自动完成这些工作。所有中断没有被禁止的情况下，不要调用 HWI\_enter。

#### ➤ HWI\_disable 和 HWI\_restore

在 C54x 上，HWI\_disable 的汇编 API 和 C 语言 API 有些许差别。C 语言的 HWI\_disable 总是返回禁止中断前 ST1 寄存器的数值。ST1 寄存器在位 11 里保存了全局中断开启位 (INTM 位)。汇编的 HWI\_disable 用一个可选参数决定是否返回 ST1 的数值，以及要存放的位置。如果调用这个 API 没有参数，就会禁止中断，不保存 ST1 的值。如果有参数调用这个



API, 则在参数指定的地方返回 ST1 的值。该参数可以是下列的数值:

A(旧数值返回到累加寄存器 A 中);  
B(旧数值返回到累加寄存器 B 中);  
\* arx(旧数值存储在 arx 寄存器指向的内存地址)。

HWI\_restore 使用相同的参数。没有参数的使用 HWI\_restore, 就是假设该数值在累加寄存器 A 中。在恢复操作中, 只有 1 位是重要的, 即第 11 位, 是 ST1 的 INTM 位。调用恢复操作时, 会根据旧的 ST1 值的第 11 位的值设置 INTM 位, ST1 的其他位不会受影响。

#### 5) 关于 C6x

对于这个处理器族, HWI\_restore 比 HWI\_enable 的代码大小更有优势。

**HWI\_enter 和 HWI\_exit 的前置条件如下:**

HWI\_enter 的后置条件就是 HWI\_exit 的前置条件。HWI\_exit 的前置条件是:

b14 = .bss 的开始指针 (b14 是数据页指针)  
amr = 0 (amr 是寻址模式寄存器)

DSP/BIOS API 的前置条件部分会详细讨论这些前置条件的含义。

HWI\_enter 的唯一前置条件是必须禁止所有中断。在跳转到 ISR 之前, 响应该中断时, DSP 会自动完成这项工作。中断没有被全部禁止的情况下, 不要调用 HWI\_enter。

#### 6) HWI 调度器

HWI 调度器提供了与 HWI\_enter 和 HWI\_exit 相同的功能。调度器是可选的, 可以从 HWI 模块属性菜单选择。若 ISR 要使用它, 可以在属性菜单上找到 HWI\_enter 和 HWI\_exit 的参数。这是编写 HWI 处理程序 ISR 的推荐方法, 因为这样就能只关注处理函数、减少代码空间和提高代码的可读性。

缺省状态下, 调度器是被关闭的, 以维护向下兼容。关闭调度器时, 调用 HWI\_enter 和 HWI\_exit 的 HWI ISR 会正常运行。但是, 启动调度器时, 必须从代码中删除所有 HWI\_enter 和 HWI\_exit 的调用。

**注意:** 启动调度器时, 使用 HWI\_enter 和 HWI\_exit 调用会让应用程序死机。

### 4. 内存

#### 1) 访问范围外的内存地址

如果你熟悉嵌入式微处理器或者任何其他 CPU, 需要知道的最重要的事就是, 在 DSP 上没有总线/寻址状态。第二件重要的事是在 DSP 上没有非法指令状态。

因此, 你能访问不存在的内存, 能从代码空间外取指令, 能执行任何当作指令取出来的垃圾命令。DSP 只会保持运行, 产生不可预知的完全相反的结果。这使得调试 DSP 程序变得非常困难。你如何处理这种当发生基本错误时并不报错的情况呢?

在硬件相关部分提供了一些建议的方法。这些方法对于代码空间外的内存访问处理得很好。但是, 对于访问不存在的内存, 逻辑分析器是一个很棒的工具。CCS 还提供了仿真分析工具, 其部分功能和逻辑分析器一样。

#### 2) 仿真分析工具

仿真分析工具是 CCS 环境的插件, 帮助你设定硬件断点和统计事件。在软件断点有问题的环境(如基于只读存储器 ROM 的代码)中, 硬件断点是非常有用的。该分析工具能用

于监控程序总线、数据总线或 I/O 总线，并且能在不同类型的总线周期上中断执行。可以针对不同的处理器族，根据特定 DSP 的能力和特性，自定义这个工具。可以用这个工具为断点设置复杂的条件。大部分调试器对软件断点可用的功能也都能对硬件中断可用（步跟踪执行、运行等）。

### 3) 关于 C54x

#### ➤ 填充内存以产生中断的指令

检测 CPU 递增程序的计数器时是否超过合法代码区域的一个方法是，用一条填充内存的指令来产生 DSP 的中断。还要在已选择的指定中断向量上设置一个断点。之后，下载应用程序代码并运行。如果处理器试图执行区域外的指令，就会产生中断，跳转到其向量，遇上断点。机器的状态（包括堆栈）可用于跟踪和分析。

CCS 里，用一个单字来填充内存。因此，在需要单字指令产生软件中断的 DSP 上，这个方法都是可行的。C54x 就是如此。

有两种不同的指令可以实现这个方法：INTR 和 TRAP。

INTR 指令的语法是：INTR < vector # >。其操作码是 0xF7Cx，其中 x 是中断向量号。例如，INTR 5 的操作码是 0xF7C5。

TRAP 指令的语法是：TRAP < vector # >。操作码是 0xF4Cx，其中 x 是中断向量号。例如，TRAP 2 的操作码是 0xF4C2。

这两条指令的区别在于，TRAP 不能禁止中断；INTR 能禁止中断。但是，这两条指令产生的中断都不是可屏蔽的，它们会产生中断，无论是否允许中断。

#### ➤ 近寻址与远寻址

最初的 C54x 设备是 16 - 位寻址的设备，因此，最大寻址空间是 64K 字。其较新的设备采用了较宽的寻址方式，可以适用于更大规模的应用。但是，增加了代码量能力的同时也增加了复杂度。

尽管内存模型只有近寻址和远寻址，但是，实际上有三种应用程序可以选择的内存模型。这几种选择也各有利弊，并且适合不同规模的应用程序。

如前所述，近寻址模型只提供 64KB 的程序内存。远寻址模型有两种：清空 OVLY 位的远寻址和设置 OVLY 位的远寻址。OVLY = 1 的远寻址模型处理 4.03MB 唯一程序空间。OVLY = 0 的远寻址模型处理 8MB 唯一程序空间，即 C54x 设备支持的最大程序空间。事实上，这是理论最大空间，并不是实际的最大空间。后面部分会讨论。处理器的 PMST 寄存器中存放了 OVLY 位。该寄存器控制了如何扩展内存和片上内存如何映射到处理器空间。

因为最初的 C54x 设备寻址空间最大为 64KB，所以，在新设备中，超过的内存被称为扩展的内存。扩展的内存按 64KB 分为多个内存页。最多支持 128 个扩展页。C54x 的 PC 总是标记一个地址在 0 ~ FFFF 区域（64KB），在一个额外的 XPC 寄存器里知道页号。近寻址和远寻址的基本区别在于，对 XPC 寄存器的使用方式。远寻址修改并使用这个寄存器；近寻址使用但并不修改它。支持远寻址的处理器有特别的指令在寻址计算中使用 XPC。

具体应用时，如需其他信息，请参阅 SPRA599 和 SPRA492。

#### ➤ OVLY = 1 时

这个设置使每个扩展内存页上的 32KB 内存（0 ~ 0x7fff）是重叠的。如果每个页是 64KB，就重叠每页的首 32KB，为每页剩 32KB 唯一存储空间，即 4MB。重叠页是额外的 32KB，所以程序空间共能使用的存储空间是 4.03MB。

这就大量丢失了潜在的寻址空间，事实是 50%。但是，这样做是有好处的。当中断发生时，处理器对向量表进行近寻址，例如，处理器假设向量表被存放在当前页。如果向量表存放在其他页，处理器会在当前页上跳转到向量的偏移，执行错误代码。这个问题的解决方法是让向量表在每页上可见。然后，当处理器跳转到向量表，它总能执行有效的向量代码。这 32K 的重叠内存提供了实现这个方案的机制。

中断向量表是一个 128 字的表，通常从 0xFF80 开始，即首 64KB 内存页的顶部。如果设置 **OVLY = 1**，向量表必须从这个地址移到扩展页上。向量表必须在首 32KB 里在一个 128B 边界上对齐。片上内存的首 128 字 (0x0 ~ 0x7f) 保存了内存映射的寄存器，不能用于有效的程序空间。因此，**可用于向量表的范围是 0x80 ~ 0x7F80**。如果你不做这个改变，向量表只会在页 0 里存在。在其他页上运行期间，如果发生中断，处理器会跳转到它认为向量表存在的地方，就会发生不可预知的结果。

在重叠页里的任何片上内存会映射到数据空间或程序空间。片上内存是应用程序可用的最快内存。最关键的算法和/或数据应该放置在这些内存里。

当 **OVLY = 1** 时，有一些其他的放置代码的限制条件。用来访问 DSP/BIOS 的 C 封装器也必须放在重叠页里，因为 DSP/BIOS 的内核对于 C 封装器和其使用的函数链接桩必须是可近调用的，即在同一个 64KB 页里。因此，DSP/BIOS 和函数链接桩必须防止在重叠页（页 0）里。

ISR 可以放在任何内存页里，但是也有一个约束条件。如果把它们放在任何内存页而不是重叠页里，HWI\_exit 或远程返回指令会立刻结束 ISR。如果应用程序使用远内存访问模型，HWI\_exit 自动执行远程返回。如果在这两个方法中，ISR 并不结束，结果就是不可预知的。这也是要使用 HWI\_enter 和 HWI\_exit 的另一个理由。

根据对片上内存和重叠空间的需求，这些约束条件会影响在内存里划分应用程序的方式。

#### ➤ **OVLY = 0 时**

此时，特别需要注意的是，使用 DSP/BIOS 配置工具时，不能允许 **OVLY = 0** 的远访问模式。进行所有远访问模式的配置时，必须 **OVLY = 1**。

在这种情况下，使用扩展的内存，且没有重叠页。片上内存也不映射到程序空间中。理论上，这就提供了 8M 的可用寻址范围，但似乎并非如此。

没有像 **OVLY = 1** 时的自动重复存储。但并不会因此而没有重复的需要，这就意味着必须手动进行内存复制。

必须拷贝中断向量表到每页上，才能在运行过程中开启中断。如果一个内存页上没有中断向量表，那么在该页上运行时就必须禁止中断。如果并没有如此操作，并且发生了中断，结果就是不可预知的。

此外，由于 C 封装器和函数链接对于 DSP/BIOS 必须是可近调用的，必须在相同页上一起复制这三个组件。如果某个页包含的代码并不调用 DSP/BIOS，就没必要在这里复制这三个组件。

但是，需要确定的是有的复制是必需的，这也是为什么 8MB 是一个理论数字。考虑复制，这个数目会减小，但它仍然比 4.03MB 大。

这里也适用 ISR 远程返回的使用规则，也包括汇编时的远程支持，所以 HWI\_exit 是实现这些的最好方法。

另一个缺点是，作为用户应用程序的一部分，所需的复制必须手动完成。这需要块复制和其他需要考虑的事项，因为要拷贝的是代码空间不是数据空间。

最明显的缺点可能是程序空间里的片上内存不足。虽然片上内存对数据空间仍然可用，但是应用程序会完全在扩展内存上运行。这会影响速度，是设计者必须考虑的因素。

### ➤ 选择期望的内存模式

下面不讨论 OVLY = 0 的远访问模式，因为不支持对这种模式的配置。

为应用程序打开 DSP/BIOS 配置。右键点击全局设置 (Global Settings)，从快捷菜单中选择属性 (Properties)。

如果要使用近访问模型，设定函数调用模式 (Function Call Model) 为近模型 (near)。这样就完成了。

如果要使用远访问模型，设定函数调用模式 (Function Call Model) 为远模型 (far)，并确保 PMST (6 - 0) 的第 5 位的值是 1。关闭全局设置窗口，双击内存区管理器 (Memory Section Manager)。右键点击 VECT，编辑中断向量表的基地址属性。

在内存区管理器里，会看到两个区：EPROG0 和 EPROG1。它们是两个扩展内存页对象，用来作为启动器。可以根据需要创建多个这种对象，最多 128 个。如果 OVLY = 1，这些区的最大尺寸是 32K。右键点击内存区管理器，增加区，就完成了创建这些对象。右键点击每一个对象，编辑其属性。为每页输入合适的基地址和长度。设定每页的空间值为代码 (code)。

在工程 (project) 下拉菜单的选项 (options) 窗口的编译器 (compiler) 和汇编 (Assembler) 标签里，添加 “-mf” 标记和 “-v54x” (根据 CPU 选择，如 -v548，-v549 等) 标记。

最后，修改链接命令文件，以将希望的代码装载到相应的扩展区。

#### 4) 关于 C6x

### ➤ 分支跳转到同一个位置 (只适用于 C6x)

检测 CPU 何时已经递增到程序的计数器超过了合法代码区域的一个方法是，使用分支跳转到自身的指令来填充内存。如果停止了正常的功能单代码仍然继续运行，就停止 DSP。你可能被这种指令所吸引。这个方法不是十分安全的，是由 C6x 的流水特性而产生的。在 C6x 上，最小的循环尺寸是 6 条指令。那意味着，它会在 6 条指令上不断循环，而不是一条指令上不断循环。大多数情况下，这都是可行的。除非，6 条指令跨越了真实数据或代码的边界 (例如，3 条指令是分支跳转到自身，后面 3 条指令是代码或数据)。此时的结果是不可预知的。但大多情况，处理器会关注于空闲循环。可以跟踪堆栈决定如何在这个非法寻址区域里结束，否则，状态位会有额外的已经丢失的调试信息。

可使用的操作码是 0x00000012。

CCS 运行用一个单字填充内存。因此，在任何需要单字指令的 DSP 上，这个方法都可以产生这个分支跳转。C6x 就是这样的处理器。

### ➤ C6x 上的近寻址和远寻址

在 C6x 上有五种内存模型：一种小内存模型，四种不同的大内存模型。为应用程序选择的模型决定了如何在内存里分配 .bss 区和如何调用子程序，它们都直接影响应用程序的时间和空间有效性。.bss 区是存储全局静态数据的地方。

### ➤ 小内存模型

这种内存模型设定了 .bss 区的最大值是 32KB。运行时访问这些数据作为从数据页指针 (DP) 开始的偏移。按照惯例，寄存器 B14 中保存这个指针，并且只在初始化时设置一次。这种简单方式允许处理器使用直接寻址方式访问 .bss 区里的数据。

小内存模型也假设子程序段在调用程序段开始加/减 1MB 的范围内。这就运行处理器使



用程序计数器（PC）相关的分支跳转。

这种模型也被称为近寻址，它是没有修改编译器设定时的缺省内存模型。也是编译 DSP/BIOS 采用的模型。由于它只需一条指令就能访问 .bss 数据或调用子程序段，它是最高效的内存模型。同时，这种模型节省了内存也节省了时间，但内存大小有限。下面给出了一些近寻址的示例：

```
LDW *DP(_data_item),A0    ;_data_item 是距离 DP 的偏移
B    _function1           ;程序计数器相关的分支跳转到 function1
```

### ➤ 大内存模型

有四种大内存模型，其区别在于大的程度。

**聚合数据大内存模型：**聚合数据包括结构和数组。只能通过远寻址访问聚合数据。这意味着，聚合数据不会存放在 .bss 区，引用它的方法不同于访问 .bss 数据所使用的方法。在程序段上的 1M 限制对于函数调用仍然适用。

如果全局静态数据大于 32KB，就应该使用这个模型。注意，远寻址不适用于函数调用和 .bss 数据的访问。使用这个模型，需要在编译选项中设定标记 “-ml0”。

**函数调用大内存模型：**这个模型中，只对函数调用使用远寻址。和数据远寻址类似，执行分支前，必须在寄存器中装载目标程序段的地址。

在 .bss 数据上的 32KB 限制仍然存在。如果函数之间的距离超过 1MB，就应该使用这个模型。需要注意的是，远寻址不影响任何数据的访问。使用该模型，需要在编译选项里设定标记 “-ml1”。

**聚合数据和函数调用大内存模型：**这个模型组合前面两种模型。远寻址不影响 .bss 数据的访问。在编译选项里设定标记 “-ml2”。在 .bss 数据上的 32K 限制仍然存在。

**函数调用和所有数据大内存模型：**这个模型通过远寻址额外地访问所有数据。此时，访问 .bss 数据就和访问聚合数据的方法一样。它消除了程序临近和 .bss 尺寸的限制，但是在效率上进行了权衡。在编译选项中设定标记 “-ml3”。

### ➤ 关于 DSP/BIOS

虽然用小内存模型编译 DSP/BIOS，但无论构建应用程序使用的内存模型，还是配置工具里创建的对象并不放在 .bss 区里。相反，每个类型的对象放在不同的区里，根据对象类型命名（如 SIO\$obj、TSK\$obj 等）。需要选择在应用程序中处理的方法。如果不处理这个问题，应用程序可能运行（根据链接器的运气）也可能不能运行。更糟的是，应用程序可能运行一会儿，当按正确或错误方法来添加足够的存储或减少存储时，应用程序就会暂停。

有几个可用的选项，每个选项都有优缺点。

最直接的解决方法是，用大内存模型编译 DSP/BIOS。这自动考虑了对远对象的访问。它也允许将对象放置在所希望的任何地方。但是，这会导致存储和访问对象所需执行时间的额外开销。根据你对大内存模型（ml0、ml2 或 ml3）的选择，可以缩小这部分额外开销。

也可以为每个对象声明一个全局对象指针，只通过这些指针访问对象。虽然这个方法会使远程访问透明，但是它会产生指针所需的额外存储以及远寻址的额外开销。一个示例如下：

```
Extern    PIP_Obj inputObj;
PIP_Obj    * myInput = &inputObj;
/* 使用 myInput 访问 inputObj 对象 */
```

第三种访问方法是声明每个对象都是远对象。如下面的例子，在 `extern` 命令里引用的对象。

```
extern far PIP_Obj inputObj;
extern far SIO_Obj inStream;
extern far LOG_Obj trace;
```

这可能是最容易实现的方法，因为只是要访问的对象是远寻址的，应用程序的剩余部分可以使用小内存模型。

最后一种方法，相比其他方法而言开销最小。在内存里，定位所有配置对象邻近于 `.bss` 区。编译器可以使用近寻址访问从 `.bss` 开始的 32K 范围内所有数据，无论这些数据在 `.bss` 区还是不在 `.bss` 区。因此，如果 `.bss` 区足够小，就可以指引连接器将配置对象直接放在 `.bss` 后面，从而消减远寻址的开销。其缺点是，必须了解 `.bss` 区和开发所需的配置对象的大小，一旦 `.bss` 和配置对象的总和超过 32K，就要改变所使用的方法。下面给出了放置配置对象的实现方法：

- 在 MEM 管理器里，定义内存段保存这些对象和 `.bss` 数据；
  - 在每个对象的属性页上，输入上述内存段的名称，将对象分配给该段；
  - 在 MEM 管理器的属性页上，为 `.bss` 区输入同一个内存段的名称。
- 推荐的方法是，声明配置对象为远寻址的，但是最后的选择取决于特定的情况。

#### ➤ 重启向量（只适用于 C6x）

在 C6x 上，调试期间有时会看到一个奇怪的行为，应用程序有时会不断地重启。这通常是可见的，因为主函数会不断地重复运行。重启的原因是程序计数器跳到了地址 `0x00000000`。这是重启向量的位置，所以跳到这个地址就会导致重启一个全新的系统。

在通常的情况下，程序计数器不应该在程序中间跳到 0。可能有几种原因导致发生这种行为。例如，使用空指针作为程序段指针时、从有坏数据或被破坏的表里检索得到一个指针时、子程序返回时堆栈被破坏，都会发生这种情况。

有的应用程序在重启之前运行无错，重启后，代码按照之前的相同路径运行，错误再次发生，即再次重启。这是一个可重现的错误模式。

C6x 是唯一会出现这种情况的处理器，因为其重启向量的位置。但是，毫无疑问，那些类似的微处理器在摩托罗拉 68K 族上也会出现同样的现象，因为它们的重启向量也在地址 0。

解决这个问题一个保护措施是，在重启向量代码上设置断点。如果问题发生，就能在发生时捕获到。

## 5. DSP/BIOS API 的前置条件

使用 DSP/BIOS 时，要遵守的最重要规则之一是，**要符合用户手册里为每个 DSP/BIOS API 调用设定的前置条件**。这些前置条件规定了调用 API 前所需的特定状态和取值。这些前置条件是必需的，如果不符合它们，API 的调用可能就不会按所希望那样运行，其结果是不可预知的。手册也给出了后置条件，说明了 API 调用结束后，特定状态和取值。

符合前置条件并不意味着必须设置所有需要的取值和状态，因为根据当前处理器状态，可能已经符合了前置条件。但是，它们也可能不符合，此时就需要设置它们。

如果一个 API 调用并没有如期望那样运作，就需要怀疑前置条件，可以通过 LOG 函数记录它们的日志。注意不要忘记，LOG 函数也是 API，也有自己的前置条件。



在不同的处理器上，API 的前置条件有很大的不同。更多细节参见相关 DSP 体系结构手册。请参考 DSP/BIOS 用户手册，查阅每个 DSP/BIOS API 的前置条件的完整说明。

## 6. 断言的使用

在代码实时运行过程中，断言是错误发生时捕获错误的工具。它们可用于函数开始时检查输入参数或函数结束时检查输出参数。许多时候，考虑时间和资源负载，在实时 DSP 应用中不进行边界检查。断言（ASSERT）允许编程人员在调试模式中进行边界检查，而在最终产品中移除。谨记，断言会在应用程序中添加代码，它们也会影响计时，所以移除它们后，应用程序必须重新校验。断言宏可以被定义完成多件事情，写日志，甚至停止 CPU。下面给出了这样一个宏的一般形式。

```
#ifdef DEBUG
#define ASSERT(a) if(!a) { <action to take> }
#else
#define ASSERT(a)
#endif
```

这是错误发生时一个非常有用的捕获错误的简单形式。“a”是宏的参数。在代码中，用设定的条件替代。如果条件测试为 FALSE，就会发生一些操作行为。包含了#ifdef，以便于代码中放置的所有断言可以原位保留并可简单地编译或不编译。从 CCS 的下拉菜单工程（Project），在选项（Options）窗口的标签编译器（Compiler）中选定 DEBUG 标记。该标记允许定义编译命令行上使用它。当不再需要该标记时，只需简单地从此处删除它，重编译，执行代码中就没有断言宏了。如果需要，在源代码中保留了断言宏用于以后的调试。

应该在高层头文件中放置上面的断言宏，这样它就能被所有的代码模块所访问。

安排的特定动作可以是一系列事情。你可以执行 LOG\_printf、LOG\_error、LOG\_message、计数器加 1 或其他对特殊状况有意义的动作。可以在一个断言中组合多个动作。

### 1) 关于 C54x

在 C54x 上，对于断言而言，一个非常有用的动作行为是插入断点的操作码。在断言条件测试为 FALSE 时，这会让 CPU 处理停止。其定义如下：

```
#define ASSERT(a) if(!a) { asm(“ . word 0xf4f0”); }
```

如果断言条件测试为 FALSE，这个代码会在执行流里插入一个断点操作码 0xf4f0。如果测试结果为 TRUE，就会按正常情况一样继续执行。停止 CPU 对于及早检测错误数值、保存机器的状态和允许跟踪问题的原因都是非常有用的。可以调用日志函数和使用断点一起进行，便于跟踪问题。

需要注意的是，虽然断点指令是单字，但是并不要企图用这个单字填充内存，而是要使用之前建议的软件中断指令。如果这样使用的话，CCS 并不能辨识这是个断点。

### 2) 关于 C6x

在 C6x 上，代码集成器不能辨识放置在断言里的断点。

## 7. 全局存储的初始化

在标准 C 中，编译器会将应用程序没有显式初始化的全局变量初始化为 0。TI DSP 编译器没有这样的功能。在运行前，必须显式地为所有全局变量赋予初值，包括希望初值是 0 的

全局变量。在 C 代码里，可以简单地在全局变量声明里指定其初值。编译器为每个数值创建一个初始化记录。在链接时，这些记录都被收集记录在 .cinit 区，生成初始化表。在装载时或运行时，在名为 autoinitialization 的进程中，使用该表初始化系统全局变量。

在 CCS 环境中，运行时自动初始化（Runtime autoinitialization）也被称为 ROM 自动初始化。在初始化时，根装载器（bootloader）拷贝 .cinit 记录到目标内存里。根装载器根据该表运行，执行指定的初始化。如果系统最终从 ROM 运行，上述方法就是系统所需的自动初始化的方式，因为系统必须要包含初始化数据。包含初始化进程的系统，重启需要花费的时间较长。

装载时自动初始化在 CCS 环境中也被称为 RAM 自动初始化。它并不将 .cinit 记录拷贝到目标内存里，相反，主机上的装载器在空闲时执行初始化。使用这种自动初始化，重启目标机器时，并不会重新初始化全局数据，只有在装载程序时才进行初始化。因此，重启能迅速完成。

在工程 Project 下拉菜单的选项 Option 窗口中，在链接器 Linker 标签页中选择自动初始化类型。

生成 .cinit 记录是 C 编译器的功能，并不是汇编器的功能。这意味着，如果在 C 模块中声明全局变量，编译器会生成 .cinit 记录；如果在汇编模块中声明全局变量，则必须自己生成 .cinit 记录。在处理器相关的部分会给出 .cinit 记录的格式。

如果不仔细按照这些格式生成记录，应用程序初始化时可能会产生错误。如果记录总数和数据值的数目不严格相等，程序可能永远不会执行 main() 函数。如果应用程序不执行 main() 函数，需要检查你的 .cinit 记录，可能需要单步跟踪初始化过程来调试这个问题。

有几种判定 .cinit 区地址的方法。一种方法是打开一个 DOS 窗口，cd 到你的工程目录，输入命名 sectti，这会列出应用程序的每个区及开始地址和大小。如果想要获得应用程序的更多细节，可以在工程 Project 下拉菜单的选项 Option 窗口的链接器 Linker 标签页中，生成一个映射文件，并输入该映射文件的名称。重新编译应用程序，然后编辑该映射文件。一旦已知 .cinit 区的地址，就可以使用 CCS 里的内存 memory 窗口检查其记录。

#### 1) 关于 C54x

在 C54x 上，.cinit 记录的格式中，首字是 MAU 中初始化数据的大小。如果该数为正数，记录的格式如下：

```
初始化数据的大小(MAU 中);  
初始化数据要拷贝到的地址;  
初始化数据。
```

关于该格式的细节信息和 .cinit 记录的示例，请参考 C54 族的优化 C 编译器的用户指南。

如果要在 C54x 上单步跟踪 .cinit 初始化的代码，该代码在程序段 \_c\_int00 中。该程序段的源代码在子目录 /ti/c5400/bios/src/misc 下的 boot.s54 文件中。

#### 2) 关于 C6x

在 C6x 上，.cinit 记录有两种可行的格式：一种是记录初始化数据值，一种是记录初始化指针。记录的首字是 MAU 中初始数据的大小。如果该数为正数，该记录的格式如下：

```
初始化数据的大小(MAU 中);  
初始化数据要拷贝到的地址;  
初始化数据。
```

对于 C6x, 如果记录的首字是负数, 其格式如下:

寄存器 DP 补充的标记;  
需要补充的地址。

第二种格式用于变量需要指向 .bss 区的其他变量的情况。由于 .bss 从 DP 的取值开始, 在 C6x 上, 只为 DP 赋值一次, 该列表里的所有地址都是 .bss 的地址, 其取值需要和 DP 相加所得。

C6x 上, .cinit 记录有额外的需求, 它们必须对齐 8 字节边界。

关于格式的详细信息和 .cinit 记录的示例, 参考每个 CPU 族的优化 C 编译器的用户指南。

### 8. DSP/BIOS API 的上下文敏感

本节前面的中断部分提到 DSP/BIOS API 的上下文敏感。除了 HWI 外, SWI 和任务也是 API 的敏感因素。DSP/BIOS 用户指南的附录 A 详细列出了各种上下文中应该和不应该使用的 API。此外, 附录 A 还标明了哪些 API 可能会引起上下文切换, 哪些 API 不会产生上下文切换。这是决定何时使用特定 API 的一个重要因素。

关于 main() 函数的特别注意事项

传统的情况下, main() 函数的程序段是用户应用程序的开始点。在完成初始化要运行的系统后, 就执行这个程序段, 运行中的系统是完全可以操控的。但是, 在 DSP/BIOS 应用中, main() 实际是 DSP/BIOS 初始化的扩展, 它是可以由用户定制的。main() 运行时, DSP/BIOS 已经被初始化, 但是它还不能运行。因此, 调度器是不可用的, 直到 main() 返回退出, DSP/BIOS 才真正开始运行。在 DSP/BIOS 开始运行前, 中断都是被关闭的。

由于这个原因, 在 main() 函数里, 阻止调用会引起上下文切换的 API。如果调用被阻塞, main() 的执行就会停止, 等待事件不再发生。

一定不要把 main() 当作应用程序的一部分, 它实际是 DSP/BIOS 的一部分。它可以用来执行系统的部分初始化, 如硬件和数据结构的初始化。但是, 需要注意硬件初始化, 因为在 main() 运行时, 中断是被关闭的。在 main() 退出和 DSP/BIOS 开始前, 个别任务不会运行。因此, 安全的方式是让任务自己完成初始化。在使用 SWI 代替任务的系统中, 更恰当的方法是在 main() 函数里进行系统初始化。

在系统启动时, 按照下面的顺序执行:

- ① BIOS\_init      初始化 DSP/BIOS 模块;
- ② main()        如果必要, 执行特定应用的初始化;
- ③ BIOS\_start    启动调度器, 开启 DSP/BIOS 模块, 开启中断;
- ④ 运行应用程序。

一旦 BIOS\_start 退出, 中断和上下文切换就都可能放生。在 main() 函数中, 不应该发生开启中断的情况。

### 3.1.5 DSP/BIOS 线程同步

DSP/BIOS 编程从单循环程序过渡到复杂的利用实时特性的多线程应用程序, 线程需要同步访问共享资源。DSP/BIOS 提供了多种机制来同步线程。这些机制提供了一系列禁用线程类型、改变线程优先级及用信号和锁来实现互斥的方法。

用户根据需要同步的线程类型来选择机制。本节介绍了各种 DSP/BIOS 线程同步原语，以及相关问题和约束。

### 1. 线程同步的问题

DSP/BIOS 允许将应用程序组织为线程的集合，每个线程实现一个模块化的功能。线程使用操作系统的服务实现同步通信，包括信号、互斥、中断保护和改变线程优先级等。

在实时应用程序中，多线程需要访问公共资源，如队列、共享变量和列表。使用临界区没有冲突地访问这些公共资源。信号、锁和禁止线程的技术可以保护临界区。

应用程序的编程人员必须保证临界区具备下列特性。

- **安全性。**在临界区中，至多只能有一个线程。也就是说，必须互斥访问共享资源。
- **响应性。**在互斥区中，如果有多于一个线程，最终至少其中一个线程进入临界区。即，无死锁发生。

本节详细讨论了不同类型的 DSP/BIOS 线程及相关联的同步机制。这些机制分为三类：

- 阻塞线程；
- 改变线程的优先级；
- 互斥。

在讨论这些机制前，首先回顾一些和线程同步相关的内容——不同类型的 DSP/BIOS 线程提供的优先级和临界区中会出现的问题。

#### 1) 线程优先级和抢占

DSP/BIOS 支持几种类型的程序线程，并赋予不同的优先级。不同类型的线程有不同的执行和抢占特性。图 3.17 按照优先级从高到低的顺序列出了不同类型的 DSP/BIOS 线程。

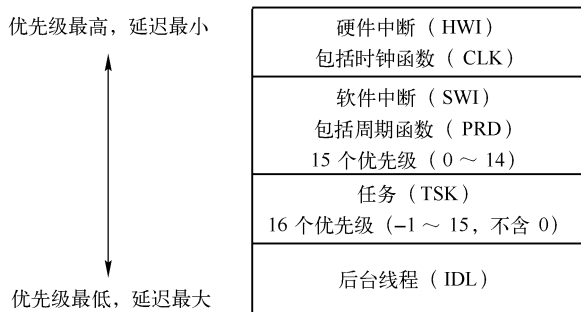


图 3.17 DSP/BIOS 线程优先级

按照优先级从高到低，线程类型包括以下几种。

- 硬件中断（HWI），包括 CLK 函数。
  - 最高优先级。
  - 一个 HWI 可以中断另一个 HWI。
- 软件中断（SWI），包括 PRD 函数。
  - 比硬件中断的优先级低。
  - 14 个子优先级。
  - 可以被较高优先级的 SWI 或 HWI 抢占。
- 任务（TSK）。

- 比软件中断的优先级低。
- 可以被较高优先级的任务抢占。
- 有 16 个子优先级。
- 如果优先级低于 0，该任务被禁止执行，直到被另一个线程提升了其优先级。优先级 0 被分配给 TSK\_idle 任务，定义位缺省的配置后台线程（IDL）。
- 后台线程。优先级最低，可以被 TSK、SWI 或 HWI 抢占。

线程优先级提供了一个保证高优先级线程的同步机制。例如，保证 HWI 线程优先于 SWI 和 TSK。类似，保证 SWI 线程优先于 TSK。也就是说，不需要保护在 HWI 线程中的临界区就可以不被 SWI/TSK 访问。

2) 竞态条件和临界区

当多线程同时竞争访问同一个资源时，应用程序中会出现竞态条件。产生竞态条件的典型原因是不同步地访问共享资源。这种情况是难以检测的。图 3.18 示例了一个在不同步访问共享变量 cnt 时出现的竞态条件。此例中，线程可以是 HWI、SWI 或 TSK。

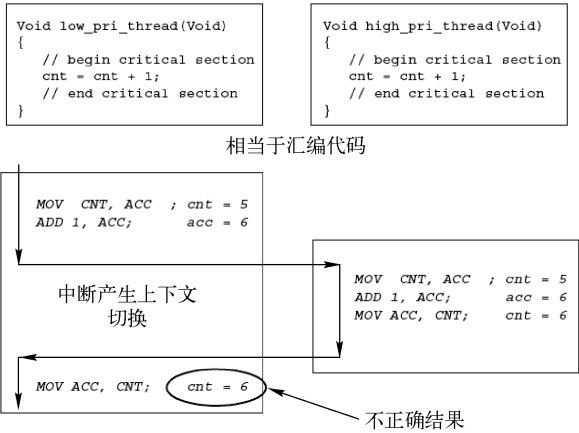


图 3.18 竞态条件的示例说明

图 3.18 中的例子产生了一个错误结果。变量 `cnt` 的初始值是 5，两个线程递增它的值后，取值应该是 7。但是，线程抢占导致最后 `cnt` 的值是 6 而不是 7。

为了防止出现这种情况，需要不可中断地执行访问共享资源的操作。这种对共享资源的原子访问就构成了临界区。这个例子中，三条递增 `cnt` 的指令应该当作临界区来处理。

图 3.18 用一个简单的变量 `cnt` 说明了竞态条件。在实时系统中，使用队列和共享数据均作为共享资源。

不同线程间会发生上下文切换。要实现临界区，需要使用一些同步原语。同步线程时，必须注意相互依赖性，还需要注意对操作系统同步服务的部署，防止它们产生死锁、高中断延迟等其他问题。

2. 阻塞线程

要保护多线程应用程序里的临界区，可以阻塞除了当前执行线程外的其他所有线程进入临界区。这保护了运行中的线程在操作临界区时不被其他线程抢占。完成对临界区的操作后，唤醒所有线程。可以调用 DSP/BIOS 的 API 实现对线程的唤醒和阻塞。

表 3.2 列举了用于阻塞各种类型线程的 API 调用。



表 3.2 调用阻塞 API 时各线程的状态

	硬件中断 (HWI)	软件中断 (SWI)	任务 (TSK)
HWI_disable	阻塞	阻塞	阻塞
SWI_disable	运行	阻塞	阻塞
TSK_disable	运行	运行	阻塞

表 3.3 列出调用每个 API 所需的 CPU 周期数、调用的上下文和其他信息。

表 3.3 开启/阻塞 API

	C62x 周期	调用上下文	备 注
HWI_disable/enable	12/12 (24)	HWI/SWI/TSK	对于短临界区最合适。快和较少的周期
SWI_disable/enable	24/64 (88)	SWI/TSK	支持嵌套调用
TSK_disable/enable	64/104 (168)	TSK	支持嵌套调研

#### 1) 屏蔽和开启 HWI

如前所述，优先级较高的线程能抢占优先级较低的线程。阻塞硬件中断 (HWI) 线程保护了所有线程不被 HWI 抢占。

在进入临界区前，一个线程可以调用 DSP/BIOS 的 API 来阻塞所有硬件中断。完成对临界区的操作后，重新唤醒 HWI。可以使用下列 API 禁止和唤醒硬件中断。

- **HWI\_disable**: 全局禁止中断。在 C6000 平台上，HWI\_disable 对应控制状态寄存器 CSR 里的 GIE 位。在 C5000 和 C2800 平台上，HWI\_disable 设置 ST1 寄存器里的 INTM 位。在这两种平台上，该指令包含 CPU 不触发任何可掩码的硬件中断。
- **HWI\_enable**: 在 C6000 平台上重新开启 GIE 位，或清空 C5000 和 C2800 平台上 ST1 寄存器里的 INTM 位。
- **HWI\_restore**: 重设值为调用 HWI\_disable 前已存在的状态。

下面代码示范了如何使用 HWI\_disable 和 HWI\_restore。禁止中断前，将当前重点状态信息存储在 key 中，用于后面的重置指令。

```
key = HWI_disable();
cnt = cnt + 1;
HWI_restore(key);
```

#### 注意事项

使用 HWI\_disable、HWI\_enable 和 HWI\_restore 时，有下列必须注意的问题。

- 各类线程 (HWI、SWI 和 TSK) 都能调用 HWI\_disable 和 HWI\_enable/HWI\_restore。
- 在禁止 HWI 期间，不能调用执行调度程序的 DSP/BIOS 调用，这包括发布其他线程的程序块和调用。这样的调用需要开启 HWI 中断或无模块超时。

下面的代码示范了如果在禁止终端的临界区里调用 SWI\_post 或 SEM\_post 会产生的问题。

```
key = HWI_disable();
cnt = cnt + 1;
SWI_post(&SWI0);
HWI_restore(key);
```



SWI\_post 的前置条件是, 如果在 ISR 上下文外调用, 则硬件中断必须是开启的 (即 `intm = 0` 或 `GIE = 1`)。这个示例中, 由于 `HWI_disable` 设置了 `intm` 或清空了 `GIE`, 第三行明显违反了 this 约束条件。

在临界区里调用 `SWI_post` 或 `SEM_post` 还会使另一个线程在禁止中断的情况下运行。因为中断会被禁止很长时间, 如果那个线程是一个 TSK, 这就特别不合理了。

- `HWI_enable` 会不考虑之前的上下文而开启中断。例如:

```
HWI_disable();
cnt = cnt + 1;
HWI_enable();
```

在这个示例中, 在调用 `HWI_disable` 之前, 中断可能已经被关闭。然而, 调用 `HWI_enable` 会开启中断而不考虑之前的状态。因此, 建议使用 `HWI_restore` 代替 `HWI_enable`, 因为 `HWI_restore` 会将 `intm/GIE` 的值重置为调用 `HWI_disable` 前的状态。

- 不支持嵌套调用 `HWI_disable` 和 `HWI_enable`。如前所述, `HWI_enable` 不会考虑调用前的状态开启中断, 因此嵌套调用 `HWI_disable/HWI_enable` 是不可能的。
- 禁止中断会直接影响中断延迟。建议将临界区设置得尽可能短, 以避免对其他中断服务的长时间延迟。

## 2) 使用 IER 和 IMR 掩码

中断开启寄存器 IER 决定了 CPU 是否响应中断。设定特定的掩码可以在操作临界区时禁止其他的 HWI。修改中断开启寄存器里特定定位可以操作 IER 掩码, 而不是通过设置 `intm` 或清空 `GIE` 来全局地影响所有中断。

可以为中断单独地设置掩码。在 C54x 目标机上, 如 C5402, 计时器中断 TINT 的掩码定义如下。

```
#define TINTMASK 0x0008
```

定义 IMR 掩码后, 调用 `C54_disableIMR/C54_enableIMR` 可以保护临界区。下例说明了这两个 API 的用法。

```
oldmask = C54_disableIMR(mask);
cnt = cnt + 1;
```

不同的 ISA 使用不同的掩码函数, 见表 3.4。

表 3.4 不同 ISA 使用的中断掩码函数

	TMS320C54x	TMS320C55x	TMS320C28x	TMS320C62x
禁止相应的中断	<code>C54_disableIMR(mask)</code>	<code>C55_disableIER0(mask)</code> <code>C55_disableIER1(mask)</code>	<code>C28_disableIER(mask)</code>	<code>C62_disableIER(mask)/</code> <code>C64_disableIER(mask)</code>
开启相应的终端	<code>C54_enableIMR(oldmask)</code>	<code>C55_enableIER0(oldmask)</code> <code>C55_enableIER1(oldmask)</code>	<code>C28_enableIER(oldmask)</code>	<code>C62_enableIER(oldmask)/</code> <code>C64_enableIER(oldmask)</code>

## 注意事项

调用 `Cxx_disableIER` 和 `Cxx_enableIER` 时, 必须考虑以下事项。

- 各类线程（HWI、SWI 和 TSK）都能调用 Cxx\_disableIER 和 Cxx\_enableIER。
- 如果使用这些 API，需要了解线程间的内部依赖关系。例如，假设任务 T1 调用 C54\_disableIMR 和 C54\_enableMR，且 T1 和另一个 TSK 线程 T2 共享变量 cnt。在 T1 里，需要屏蔽调用或唤醒 T2 的中断以同步访问 cnt。这防止了在临界区内切换到 T2，以实现原子访问共享变量。
- 禁止中断时，必须禁止会上下文切换到访问共享资源的线程的 DSP/BIOS 调用。在 Cxx\_disableIER/Cxx\_enableIER 块中，调用 SWI\_post、SEM\_post、TSK\_yield 或让其他线程运行的 API 会违反临界区的约束条件。如下面的例子：

```
oldmask = C54_disableIMR( mask );
cnt = cnt + 1;
SWI_post( &SWI0 );
C54_enableIMR( mask );
```

在这个例子里，SWI\_post 会上下文切换到 SWI0。如果 SWI0 也共享 cnt，就会产生对这个共享变量的非同步访问。禁止中断时调用这个调度 API 的另一个问题是，该调度 API 会让线程在禁止中断的情况下运行。由于会长时间禁止中断，这显然对于一个 TSK 线程是不合适的。

- 使用 Cxx\_disableIER 禁止所选择的中断，并允许其他中断发生。然而，如果在操作临界区时发生了另一个中断，可能会切换到一个 SWI 或 TSK。你可以使用 SWI\_disable 和 SWI\_enable 避免这种情况。在调用 SWI\_enable 之后，上下文切换才会发生。
- 建议临界区尽可能小，以防止服务其他中断的长延迟。

### 3) 使用 HWI\_enter/HWI\_exit 和 HWI 分派器

可以使用 HWI\_enter/HWI\_exit 宏或 HWI 分派器的 IER/IMR 掩码来操作 HWI 线程间的逻辑优先级。它们的概要描述如下。

- **HWI\_enter**：用来保存 DSP/BIOS 终端服务程序 ISR 的相关上下文的 API（汇编宏）。
- **HWI\_exit**：用调用 DSP/BIOS 中断服务程序前的信息恢复上下文的 API（汇编宏）。
- **HWI 分派器**：提供了一种简单的方法在 C 语言中写 ISR，应用程序负责保存和恢复上下文。

如“使用 IER 和 IMR 掩码”所述，在执行处理临界区代码时，可以设定 IER/IMR 掩码来屏蔽其他 HWI。

下面的代码使用了 C54x 目标机上的 HWI\_enter 和 HWI\_exit 宏来操作 HWI 线程间的逻辑优先级。

```
HWI_enter MASK,IMRDISABLEMASK
;isr 代码
```

假设，ISR1 是调用 HWI\_enter 和 HWI\_exit 的 HWI 线程。ISR1 与 ISR2 共享 cnt，使用 HWI\_enter 的 IMRDISABLEMASK 屏蔽 ISR2。这就防止了在访问临界区时切换到 ISR2。因此，对该共享变量的访问是原子的。

重要的是，必须在所有 ISR 代码的开始位置使用 HWI\_enter。不能使用这两条指令只保护访问临界区的 ISR 代码。在 ISR 的末尾，使用 HWI\_exit 宏的 IMRRESTOREMASK 来重新

启动中断。

可以使用配置文件（.cdb）里的调度器配置 HWI 对象。在配置文件里，可以指定 HWI 对象属性的中断掩码（IER/IMR）来实现。事实上，HWI 调度器为 HWI 对象调用包含成对 HWI\_enter/HWI\_exit 宏的函数，即调用函数前为要屏蔽的中断设置掩码，退出函数前重启中断。

### 注意事项

调用 HWI\_enter 和 HWI\_exit 宏或使用 HWI 调度器时，必须注意以下事项。

- 只有 HWI 线程才能调用 HWI\_enter/HWI\_exit 宏或使用 HWI 分派器。
- 如果使用这些 API，你需要了解线程间的内部依赖关系。特别地，需要知道 ISR 可能访问临界区的所有中断，并使用 HWI\_enter/HWI\_exit 宏的 IER/IMR 掩码或在 HWI 调度器配置来屏蔽这些中断。
- 不能使用 HWI\_enter 和 HWI\_exit 只保护 ISR 的一部分代码。下面的代码不会正确完成，甚至可能导致致命错误，因为在一个 ISR 里嵌套使用 HWI\_enter/HWI\_exit 是不可能的。

```
HWI_enter MASK,IMRDISABLEMASK  
;isr 代码的剩余部分  
HWI_enter MASK,IMRDISABLEMASK  
; 临界区代码  
HWI_exit MASK,IMRRESTOREMASK
```

- HWI\_enter/HWI\_exit 宏和 HWI 调度器屏蔽了被选的中断，且允许其他中断发生。如果在保护区内，发生了另一个中断，而该中断可能会上下文切换到一个 SWI 或 TSK。可以为整个被保护区域使用 SWI\_disable 和 SWI\_enable，防止这种上下文切换。这样，调用 SWI\_enable 后，才完成这个切换。
- 屏蔽中断的直接结果是影响中断的反应时间。推荐的方法是，把临界区设置得尽可能小，以避免服务其他终端的长时间延迟。

### 4) 屏蔽和开启 SWI

“屏蔽和开启 HWI”说明了屏蔽 HWI 线程以保护临界区的方法。屏蔽线程的原理也能应用于 SWI 线程。可以使用 SWI\_disable 和 SWI\_enable 防止被 SWI 线程抢占。SWI\_disable 禁止所有其他 SWI 线程运行，直到调用 SWI\_enable。HWI 线程依然可以运行。

调用 SWI\_disable 和 SWI\_enable 来保护临界区的方法类似于调用 HWI\_disable/HWI\_enable。如下面的例子：

```
SWI_disable( ) ;  
cnt = cnt + 1 ;
```

### 注意事项

调用 SWI\_disable 和 SWI\_enable 时需要注意以下几点。

- HWI 线程不能调用 SWI\_disable 或 SWI\_enable。
- 调用 SWI\_disable/SWI\_enable 不能保护访问临界区时不被 HWI 抢占。
- 可以嵌套调用 SWI\_disable 和 SWI\_enable。SWI\_disable 和 SWI\_enable 的调用数目必须

相同。只有最外围的 SWI\_enable 调用才实际开启软件中断。

- 在 SWI\_disable/SWI\_enable 块中，需要调度的调用（如 SEM\_post 或 TSK\_yield）只有在 SWI\_enable 之后才执行。
- 如果信号量是不可用的，在 SWI\_disable/SWI\_enable 块中调用 SEM\_pend 会立即返回 FALSE，即使超时值非零或 SYS\_FOREVER。

#### 5) 阻塞和运行 TSK

和 HWI 与 SWI 线程类似，可以使用 TSK\_disable 和 TSK\_enable 调用保护临界区。TSK\_disable 关闭了 DSP/BIOS 的任务调度器。当前任务继续执行直到调用 TSK\_enable。

TSK\_disable 和 TSK\_enable 调用的使用方法类似于 HWI/SWI 的屏蔽和开启调用，如下例：

```
TSK_disable();
cnt = cnt + 1;
TSK_enable();
```

#### 注意事项

调用 TSK\_disable 和 TSK\_enable 时需要注意以下几点。

- TSK\_disable/TSK\_enable 不能防止 HWI 和 SWI 抢占执行对临界区的访问。
- 在 TSK\_disable/TSK\_enable 块中，任何内核操作都不会阻塞当前任务，包括 SEM\_pend（除了超时值为 0）、TSK\_sleep 和 TSK\_yield。
- 不能在 SWI 或 HWI 中调用 TSK\_disable/TSK\_enable。
- 可以嵌套调研 TSK\_disable 和 TSK\_enable。不能切换 TSK，直到调用 TSK\_enable 的次数和调用 TSK\_disable 的相同。
- TSK\_disable 只能阻止较高优先级的任务运行，因为在 TSK\_disable/enable 块中关闭了调度器。这种延迟可以使用信号量来避免，参见 4.2.3.5 节“使用信号量实现共享互斥”。
- TSK\_disable/TSK\_enable 块中的调度调用（如 SEM\_post 和 TSK\_yield）只在 TSK\_enable 后才被执行。

#### 3. 改变线程的优先级

在前面，讨论了阻塞线程来实现同步的方法。下面讨论改变线程的优先级来保护临界区。这个方法是基于在运行时可以改变 SWI 和 TSK 线程的优先级的事实，来防止它们在操作临界区时被抢占。

在进入临界区前，可以将当前执行线程设置为该类型中优先级最高的线程。对临界区的操作执行完成后，将线程的优先级恢复为其以前的优先级别。

##### 1) 改变 SWI 的优先级

可以使用 SWI\_raisepri 和 SWI\_restorepri 改变 SWI 的优先级。

- **SWI\_raisepri**：提高当前运行的 SWI 的优先级到参数传递的优先级。
- **SWI\_restorepri**：将 SWI 的优先级恢复到调用 SWI\_raisepri 之前的优先级。

下例示范了如何使用 SWI\_raisepri 和 SWI\_restorepri。

```
/* 提高当前 SWI 的优先级到应用程序中最高的 SWI 优先级 */
mask = SWI_getpri(&highest_pri_swi);
```

```
key = SWI_raisepri(mask);
cnt = cnt + 1;
```

在这个例子中, 进入临界区前, SWI\_raisepri 将当前 SWI 设置为应用程序中使用的最高优先级。提升 SWI 的优先级前, 将当前的优先级保存在 key 中。完成对临界区的操作后, 用 key 的值将 SWI 的优先级恢复到原来的级别。

### 注意事项

调用 SWI\_raisepri 和 SWI\_restorepri 时, 需要注意以下几点。

- 只能在 SWI 的上下文中调用 SWI\_raisepri/SWI\_restorepri, 不能从 HWI 或 TSK 线程调用它们。
- 应用程序的生命周期内, 都可以改变 SWI 的优先级。因此, 使用这个方法实现共享互斥是危险的, 线程优先级的改变会破坏共享互斥。
- 使用 SWI\_raisepri 时, 需要知道系统里 SWI 的最高优先级。否则, 可能会把操作临界区的 SWI 提高到最高的优先级 (优先级 14)。
- 这个方法不能避免 HWI 访问临界区。HWI 线程的优先级比 SWI 线程高, 因此会破坏共享互斥。
- SWI\_raisepri 不能降低一个 SWI 的优先级。

### 2) 改变 TSK 的优先级

在上一小节中, 改变 SWI 线程的优先级能保护共享资源。这个原理也能应用于 TSK 线程, 可以使用 TSK\_setpri/TSK\_create 来实现。

- **TSK\_create**: 创建一个新任务对象, 并使其准备执行。可以指定任务的执行优先级属性。
- **TSK\_setpri**: 为一个任务设置执行优先级, 返回该任务的旧优先级。

在下面的示例里, 可以自动创建一个任务。

```
TSK_Attrs attrs;
TSK_Handle task;
attrs = TSK_ATTRS;           /* 默认的任务属性 */
```

下面的示例使用 TSK\_setpri 保护临界区。

```
Uns oldpri;

oldpri = TSK_setpri(TSK_self(), newpri);
cnt = cnt + 1;           /* 临界区 */
```

与前面对 SWI 线程的操作一样, 进入临界区前, TSK\_setpri 使当前任务是应用程序中优先级最高的任务。提升当前任务的优先级前, 将优先级掩码保存在 oldpri 里。完成对临界区的操作后, 使用 oldpri 将当前任务的优先级恢复到原来的级别。

### 注意事项

使用 TSK\_setpri 时需要注意以下几点。

- 这个方法不能避免 SWI 和 HWI 操作临界区。因此, 从线程同步的角度, 只能调用

TSK\_setpri 实现同步访问任务线程间共享的资源。

- 在临界区内，调用 SWI\_post、SEM\_post 或 TSK\_yield 会破坏临界区约束，如下例所示。

```
oldpri = TSK_setpri( TSK_self(), newpri );
cnt = cnt + 1;           /* 临界区 */
SWI_post( &SWI0 )
```

在这个例子中，调用 SWI\_post 造成上下文切换到 SWI0。如果 cnt 也共享给了 SWI0，这就构成了对共享变量 cnt 的不同步访问。因此，临界区内，不能包含会上下文切换到也共享该变量的线程的 DSP/BIOS API 调用。

- 新的优先级不能是 0。这个优先级是预留给 TSK\_idle 任务的。
- 和 SWI 线程一样，应用程序的生命周期内，都可以改变 TSK 的优先级。因此，用这个方法实现共享互斥是危险的，因为线程优先级的改变会破坏共享互斥。
- 使用 TSK\_setpri，需要知道系统中 TSK 的最高优先级。否则，你可能会把执行临界区的 TSK 提升到最高优先级（优先级 15）。

#### 4. 共享互斥

DSP/BIOS 提供了诸如信号量和锁这样的结构来支持多线程环境中的同步。下面将示范如何使用信号量和锁来实现共享互斥（简称互斥）。表 3.5 列出了执行本节中讨论的 API 调用时所需的 CPU 周期数、调用的上下文及其他信息。

表 3.5 同步 API

	C62x 中的周期数	调用上下文	备 注
SEM_pend/post	228/264 (492)	TSK、SWI、HWI	程序中会发生死锁和优先级倒置
LCK_pend/post	252/296 (548)	TSK	处理了递归死锁的问题，仍会发生优先级倒置

##### 1) 使用信号量使用共享互斥

SEM 对象是计数信号量，可用于任务同步和共享互斥。计数信号量记录了相应可用资源的数目。当计数大于 0，获得一个信号量时，不阻塞任务。

可以使用配置工具静态创建 SEM 对象，或调用 SEM\_create 动态创建。要创建一个互斥信号量，必须使用初始值 1 来创建 SEM。使用下列 API 来实现共享互斥。

- SEM\_pend**：如果信号量计数大于 0，SEM\_pend 将该计数减 1，并返回 TRUE。否则，SEM\_pend 挂起当前任务的执行，直到调用 SEM\_post 或超时过期。
- SEM\_post**：为信号量唤醒第一个等待的任务。如果没有任务等待，SEM\_post 简单地将信号量计数加 1，并返回。

下面的例子使用了 SEM\_pend 和 SEM\_post 来实现共享互斥。

```
Void task0( void)
{
    SEM_pend( mutex, SYS_FOREVER );
    ' 共享的资源/临界区
    SEM_post( mutex );
}
```

```
Void task1( void)
{
    SEM_pend( mutex, SYS_FOREVER );
    ' 共享的资源/临界区
    SEM_post( mutex );
}
```



在这个例子中，用初始计数 1 创建互斥信号量。每个任务进入临界区前调用 SEM\_pend。如果互斥计数是 1，SEM\_pend 将计数从 1 变成 0 以获取一个互斥体。如果互斥计数是 0，SEM\_pend 阻塞执行以等待互斥体。临界区操作完毕后，任务调用 SEM\_post，将计数从 0 变成 1 以释放互斥体。

### 注意事项

调用 SEM\_pend 和 SEM\_post 时，需要注意以下几点。

- 这个方法不能阻止 SWI 和 HWI 操作临界区。
- 仅当超时值为 0 时，HWI 和 SWI 线程才能调用 SEM\_pend 和 SEM\_post。
- 信号量会导致递归信号量死锁。此时，两个以上的线程直接或间接等待相同共享资源，它们由于没有访问临界区资源都不能继续执行，如下面的例子所示。

```
Void task()  
{  
    SEM_pend(mutex,SYS_FOREVER);  
    ' 访问共享资源  
    func();                               /* 调用 func */  
    SEM_post(mutex);  
}  
Void func()  
{  
    SEM_pend(mutex,SYS_FOREVER);          /* 将永远阻塞,死锁 */  
    ' 访问共享资源  
    SEM_post(mutex);  
}
```

在这个例子里，第二个 SEM\_pend 调用永远阻塞该任务。这是一个典型的示例，示范了多代码路径共享函数，该函数引起了递归调用 SEM\_pend，导致死锁。这种形式的死锁在调试时相对容易检测，但是，解决这个问题会使代码复杂。

### 2) 使用锁实现共享互斥

可以按类似 SEM 的方式使用锁（LCK）来实现共享互斥。可以使用配置工具静态创建 LCK 对象，也可以调用 LCK\_create 动态创建。使用下列 API 来实现共享互斥。

- **LCK\_pend**：获得锁的所有权，以允许当前任务独立访问相应的资源。如果另一个任务已经获得该锁的所有权，LCK\_pend 就会挂起当前任务的执行，直到这个资源可用。
- **LCK\_post**：放弃锁的所有权，唤醒第一个等待相应资源的任务（如果存在）。

如果当前任务对同一个锁多次调用 LCK\_pend，当前任务会保留其所有权，直到调用 LCK\_post 的次数和调用 LCK\_pend 的一样。可以使用 LCK 的这个属性避免前面显示的递归信号量死锁场景的发生，如下例所示。

```
Void task()  
{  
    LCK_pend(mutex,SYS_FOREVER);  
    ' 访问共享资源  
    func();                               /* 调用 func */
```

```

LCK_post( mutex );
}
Void func( )
{
/* LCK 允许所有者多次获取相同的锁。无死锁。 */
LCK_pend( mutex, SYS_FOREVER );
' 访问共享资源
LCK_post( mutex );
}

```

在这个例子中，LCK\_pend 的第二次调用允许任务获取该资源，即使相同的 LCK 已经被同一个任务所有。

### 注意事项

调用 LCK\_pend 和 LCK\_post 时，需要注意以下事项：

- 这个方法不能阻止 SWI 和 HWI 访问临界区；
- HWI 和 SWI 线程不能调用 LCK\_pend 和 LCK\_post。

### 3) 避免多互斥体死锁

如果系统中有多线程等待共享资源的互斥体，多线程间会发生多互斥体死锁。使用 LCK 和 SEM 对象，都会发生多互斥体死锁。

下面的例子给出了调用 LCK API 的多互斥体死锁。

```

Void task1( )
{
LCK_pend( lockA, SYS_FOREVER );
' 访问共享资源 A
f1( );
LCK_post( lockA );
}
Void f1( )
{
LCK_pend( lockB, SYS_FOREVER );
/* 死锁 */
' 访问共享资源 B
LCK_post( lockB );
}

```

```

Void task2( )
{
LCK_pend( lockB, SYS_FOREVER );
' 访问共享资源 B
g2( );
LCK_post( lockB );
}
Void g2( )
{
LCK_pend( lockA, SYS_FOREVER );
/* 死锁 */
' 访问共享资源 A
LCK_post( lockA );
}

```

两个线程都被彼此访问的共享资源阻塞，然后就发生一个死锁。CCS 里，可以使用内核对象视图（KOV）来检测这类死锁。

建议，使用易于理解的方式获得互斥体，可以避免多互斥体死锁。应用程序中，互斥体的数目也应该尽可能最小。简单的应用程序应该尽量只使用一个互斥体。

### 4) 避免优先级倒置

当高优先级任务必须长时间等待低优先级任务时，就发生了优先级倒置现象。图 3. 19 给出了这种场景的示例。

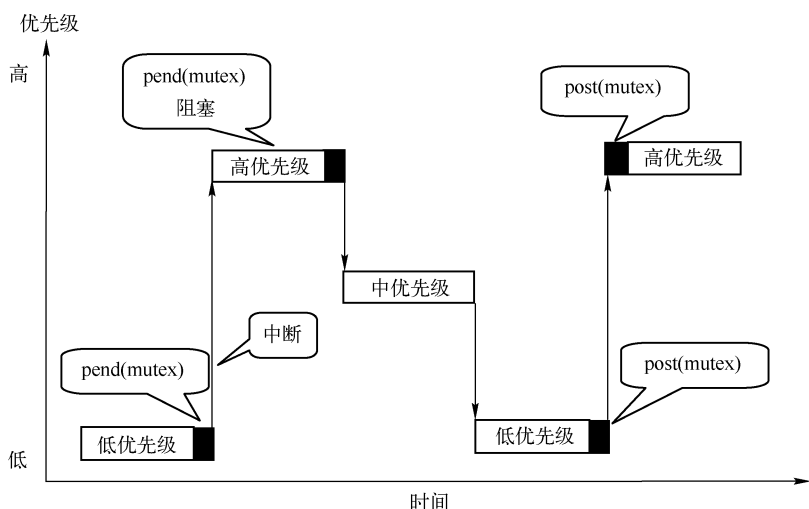


图 3.19 优先级倒置

在图 3.19 中，低优先级、中优先级和高优先级是不同优先级的任务。

低优先级任务使用互斥体信号量的 `pend` 指令获取资源。高优先级任务抢占低优先级任务，并且使用同一个互斥体信号量的 `pend` 指令竞争访问资源，高优先级任务就被阻塞。

如果高优先级任务被阻塞的时间不比低优先级任务完成对共享资源的使用时间长，就不会有问题。但是，如果低优先级任务又被中优先级线程抢占，低优先级线程就不能操作共享资源而要放弃它，这就导致高优先级任务被长时间阻塞。这样的场景会彻底地影响系统的实时行为。

在 DSP/BIOS 应用程序里，解决这种问题的方法是，临时提高低优先级任务的优先级，使其优先级是使用同一个互斥体的任务中最高的。这样，低优先级任务就能不被抢占地完成对临界区的操作，并释放互斥体，如图 3.20 所示。

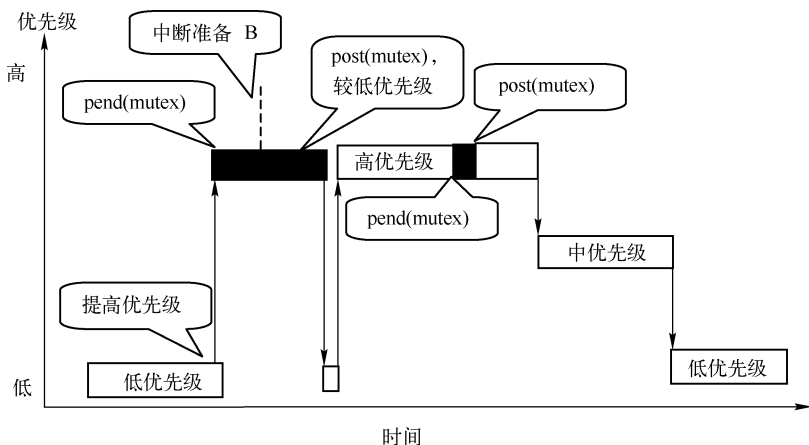


图 3.20 互斥访问前提高优先级

图 3.20 中，在低优先级任务开始访问临界区前，提高其优先级到高优先级任务的级别，这可以调用 `TSK_setpri` 实现。在低优先级任务（现在是在高优先级下运行）完成临界区的访问后，释放互斥体，其优先级会降低到它之前的状态。这样，低优先级任务可以在不被中优

先级任务抢占的情况下完成对临界区的访问。

### 5. 运行时支持 (RTS) 库的可重入问题

使用 DSP/BIOS 时, 大部分运行时支持库函数都是可重入的且线程安全的。可重入的函数是可以被不同线程调用多次且不会导致竞态条件。RTS 库函数有一个保证线程安全的内嵌机制, 且 DSP/BIOS 使用这个机制。

不使用 DSP/BIOS 时, 缺省不使用这个机制, 这时 RTS 函数是不可重入的。malloc() 是最常见的例子。stdio 函数 (fopen、fread 等) 和 printf 也是不可重入的。基本上, 当不使用 DSP/BIOS 时任何需要和 CCS 通信交互的函数都是不可重入的。此外, 有状态的函数也是不可重入的。

运行时支持库提供了 \_lock() 和 \_unlock()。DSP/BIOS 初始化这两个锁函数来支持可重入性。特别地, \_lock() 和 \_unlock() 被初始化为指向 LCK\_pend 和 LCK\_post 的指针, 它们使用了一个互斥体 LCK\_obj。重要的是, 不能使用信号量实现 lock() 和 unlock(), 因为可能会有嵌套调用。printf 是一个需要嵌套调用 lock 和 unlock 的 RTS 函数。

使用 LCK\_pend/LCK\_post 机制实现可重入性有一个约束条件, ISR 不能调用 printf。这是因为, HWI/SWI 不能调用 LCK\_pend, 否则会发生错误的调度。

### 6. 片级支持库 (CSL) 的可重入问题

在片级支持库 (CSL) 里, 所有的 \_open() 和 \_close() 函数都是线程安全和可重入的。这是因为, 它们访问的都是 IRQ\_globalDisable/IRQ\_globalRestore 块里的数据结构。可以使用这些函数的可重入性确保整个模块的可重入性。使用任何 CSL API 之前, 应用程序的代码会使用 \_open 打开外设, 验证它是否接受了一个有效的句柄。两个线程不会接受来自相同外设的句柄, 因为 \_open 函数是线程安全的。这样的轮流访问确保了两个线程不会调用相同的 CSL API 访问相同的设备。因此, 两个线程也不会试图操作相同的外设寄存器。

在两个线程间共享句柄会是一个问题。但是, 可以轻松地确保两个线程进行 CSL 调用。此外, 不是所有的 CSL 模块都是基于句柄的, 这意味着它们没有 \_open/\_close 函数。这时需要解决调用这些模块时的可重入问题。

为 DSP 应用采用一种特定的同步方法时, 重要的一点是要使应用程序的设计尽可能地简单。考虑系统里不同线程间相对的优先级, 应该避免频繁改变优先级, 这可以避免不必要的上下文切换和优先级倒置。还要尽量使 SEM 和 LCK 互斥体的数目最小, 过多使用 SEM 和 LCK 容易导致多互斥体死锁这样的问题。

选择特定的同步方法时, 需要仔细分析应用程序的实时需求。每种方法里 API 的使用会有不同的负荷。没有一种方法可以适用所有的应用程序。应根据系统的需求, 选择合适的方法。如果需要, 应用程序可以使用多种方法来完成线程同步。

## 3.1.6 DSP/BIOS 系统时钟

DSP/BIOS 需要一个心脏, 因为许多 API 都有一个超时参数, 这个心脏称为系统时钟。系统时钟的跳动和 DSP 时钟的跳动不同。除了系统时钟, DSP/BIOS 还要提供一个高分辨率和低分辨率的时间。使用这些时钟调节时间、生成时间戳消息、作为缺省心跳来驱动周期函数的执行。

DSP 常有多片上计时器来产生周期的硬件中断。DSP/BIOS 通常使用其中的一个片上

计时器作为片上系统时钟源。在缺省配置里，系统时钟使用相同的值作为低分辨率时间。没有必要用片上计时器驱动系统时钟。可以使用外部时钟或片上外设触发的 ISR 来代替计时器驱动系统。如果使用外部时钟，就需要调用 PRD\_tick() 来启动系统时钟。也可以使用片上外设触发的周期中断作为系统时钟，此时，中断的硬件 ISR 需要调用 PRD\_tick()。

### 1. 高/低分辨率的时间

片上计时器有两个寄存器，称为计时器周期寄存器 (PRD) 和计时器计数寄存器 (CNT)。高/低分辨率时间就依赖于这两个寄存器。TMS320C6711 里，CPU 时钟每滴答 4 次，计时器 CNT 寄存器就加 1。计时器 CNT 递增的频率依赖于 DSP 的生成频率，参见表 3.6。

表 3.6 DSP 的计时器时钟

属性	TMS320C620x/C670x	TMS320C621x/C671x	TMS320C64x	TMS320C54x/C55x
计时器输入时钟频率	CPU 频率/4	CPU 频率/4	CPU 频率/8	CPU 频率/(TDDR + 1)

当计时器 CNT 寄存器的取值等于计时器 PRD 寄存器的取值时，CNT 被重设为 0，并产生一个计时器中断。当计时器中断发生时，DSP/BIOS 就递增内核变量 CLK\_R\_time。调用 CLK\_gettime() 能获取低分辨率时间，它返回的是调用该 API 时已经产生的计时器中断的数目。高分辨率时间是一个更精确的值，即 CLK\_R\_time 里记录的计时器中断数与计时器周期的乘积与计时器 CNT 寄存器的取值的和，这个结果值非常接近一条指令周期。在读寄存器过程中，可能发生计时器的计数器滚转，此时，在返回高分辨率时间值前就要适当地补偿该结果值。使用低分辨率时间为长周期事件的日志添加时间戳。将高分辨率时间与 STS\_set() 和 STS\_delta() 相结合作为基准测试代码。也可以使用高分辨率时间为事件日志添加时间戳。在 C5000 上计时器周期设置为 0xFFFF，C6000 上周期设置为 0xFFFFFFFF，DSP/BIOS 使用如下优化版本的 CLK\_gettime 和 CLK\_gettime。

```
CLK_R_time = 计时器中断数
低分辨率时间 = CLK_R_time
高分辨率时间 = (CLK_R_time * 计时器 PRD) + 计时器 CNT
```

### 2. 时钟函数和操作

CLK 模块提供了四个 API：CLK\_gettime()、CLK\_gettime()、CLK\_countspms() 和 CLK\_getprd()。前面已经讨论过了前两个 API。CLK\_countspms() 返回的是已编码的每毫秒硬件计时寄存器的滴答数，CLK\_getprd() 返回的是配置的计时器 PRD 寄存器存储的值。硬件中断 14 缺省是计时器 0 的中断，DSP/BIOS 系统时钟缺省是计时器 0。

当计时器中断发生时，运行 HWI\_INT14 对应的 ISR，也就是 CLK\_F\_isr()。CLK\_F\_isr() 完成一些基本的中断服务操作，即递增 CLK\_R\_time (低分辨率时钟)，将控制转给最终返回到中断发生的上下文环境的时钟钩子函数。该钩函数被硬件中断的开始结束标志 (HWI\_enter, HWI\_exit) 包装。常用的钩函数是 CLK\_F\_run()，该函数主要就是调用函数 FXN\_F\_run()。函数 FXN\_F\_run() 按顺序调用所有的已配置的时钟函数。所以，当计时器中断发生时，在硬件 ISR 的上下文环境里执行所有的时钟函数。因此，任何 CLK 函数完成的处理操作总量最小，这些函数只调用 ISR 允许的 DSP/BIOS API。

### 3. 配置和理解 CLK 函数

下面介绍如何配置一个每 50 $\mu$ s 执行一次的时钟函数。打开配置文件 hello.cdb，右键点击 CLK - Clock Manager，选择属性。在属性窗口里，CPU 中断缺省设置为 HWI\_INT14，时钟被设置为有片上计时器 0 触发。在 microseconds/Int 域里，输入 50，指明计时器中断应该发生的时间周期。使用 microseconds/Int 域为 PRD 寄存器设置一个合适的取值，也可以选择直接配置片上计时寄存器的选项来设置计时器周期寄存器。应用这些设置，现在计时器 0 被配置为每 50 $\mu$ s 中断一次，这意味着 DSP/BIOS 系统时钟会每 50 $\mu$ s 走动一次，因此，每 50 $\mu$ s 就会执行一次已配置的 CLK 函数。将 microseconds/Int 域设定为 50 $\mu$ s 时，每次中断时，PRD 寄存器的域和指令就会自动设置一次。DSP 的 MHz 速率是此计算的基本参考。这些值的计算如下。

$$\text{Microseconds/Int} = 50\mu\text{s} = 50 \times 10^{-6}$$

$$\text{DSP 时钟滴答 1 次的时间间隔} = 1 / (150 \times 10^6) = 10^{-6} / 150$$

$$\text{PRD 寄存器递增 1 次的时间间隔} = 4 / (150 \times 10^6) = (4 \times 10^{-6}) / 150$$

$$50\mu\text{s 内 PRD 递增的次数} = (50 \times 10^{-6} \times 150) / (4 \times 10^{-6}) = 1875 = 7500 \text{ DSP 时钟}$$

再次右键点击 CLK - Clock Manager，选择插入 CLK。插入一个 CLK 对象后，你可以重命名该对象。右键点击该新创建的 CLK 对象，CLK0，将其与一个时钟函数关联，如 my\_clock()。如果是 C 编写的，需要在函数名前加下划线。应用这些设置，此时，你已经完成了对时钟函数的配置。

在文件 hello.c 里，函数 my\_clock() 的定义如下：

```
Void my_clock (Void)
{
    LOG_printf (&trace,"In clock function my_clock ()");
}
```

保存并构建 hello2 工程，装载并运行该程序，从日志窗口脚本里，你会看到重复调用函数 my\_clock()。

### 4. 基准测试的提示

#### 1) 为何要有基准提示

应用报告 DSP/BIOS II Timing Benchmarks on the TMS320C6000 DSP (SPRA66) 里提供了 DSP/BIOS 内核的性能基准测试，包括有测试的内核和无测试的内核间性能的比较。DSP/BIOS 为基准测试和剖析用户代码提供了一组 API，这些 API 由同级对象管理器 (STS) 和 CLK 模块提供，并用于剖析代码段。有些特定情况下，这些 API 可能会返回不一致或不正确的结果值。下面将介绍测试 API 返回不正确结果值的各种情况。在基准测试或剖析代码时，必须注意这些条件。

#### 2) 在有计时器的情况下运行

在应用程序里，使用计时器作为周期中断来驱动 DSP/BIOS 系统时钟，在时间不同或 DSP 不同的情况下，基准测试和剖析结果可能会不同。原因是 CPU 暂停时计时器可能仍然在运行。在 C620x/C670x 上，如果计时器时钟源被设置为内部的 CPU 时钟，CPU 暂停时计时器也暂停。这种情况下，仅当 CPU 没有被仿真驱动器中止时，计时器的计数器才被启动。此时，剖析的结果可能是合理的。在 C621x/C671x 和 TMS320C64x 里，即使在 CPU 因为仿



真驱动器停止而被停止时, 计时器仍然按照已编程的模式继续计数。这样, 如果执行停止与不确定, 获取的剖析/基准测试值也不确定。下面举例说明有时器运行时如何会给出不正确的剖析结果。假设, 用户要在要剖析的代码段的开始和结束位置设置断点。该程序运行并碰上第一个断点, 此时程序暂停, 并可能发生计时器中断, 当继续运行该程序时, 在该代码段里插入了计时器 ISR 代码。这种情况下, 获得的剖析结果值将会是代码段和 ISR 执行的时间总和。

### 3) 测试 API 的益处

实时分析是指在系统实施操作过程中获取数据进行分析。其目的是, 判断系统是否按照设计约束运行。对于实时系统而言, 周期调试对于不确定性的执行和严格的时间约束并不有效。DSP/BIOS 提供了一组测试 API 来弥补周期调试的不足。使用这些 API 比剖析器更好, 因为它们的执行时间是确定的且非常短。由于这些额外的时间是确定的, 这些 API 花费的时间是可知的, 所以可以从度量结果里将这些额外的时间分解出来。DSP/BIOS II Timing Benchmarks on the TMS320C6000 DSP (SPRA662) 给出了这些测试 API 的基准结果。

相比 CCS 的剖析工具而言, DSP/BIOS 测试 API 有一些优势。剖析器在程序执行过程中收集程序的分析数据, 而这些数据可能并不适于度量实时系统的性能。而这正是测试 API 的优点。在优先级最低的后台空闲 (IDL) 线程里, 目标机和主机间用测试 API 通信。IDL 线程只在应用程序空闲时执行。这就保证了应用程序的实时行为。测试数据总是在主机上格式化。这减轻了目标机上处理测试数据的负荷。DSP/BIOS 为实施系统提供了各种测试 API, 这些 API 比剖析器能提供更可信赖的性能数据。

### 4) CLK\_gettime()/CLK\_gettime() 的局限性

CLK\_gettime() 和 CLK\_gettime() 常用于基准测试和剖析。如果长时间屏蔽中断, 这些 API 可能会返回错误的结果值。CLK\_gettime() 返回系统的高分辨率时间。是机器设备有一个周期寄存器和计数寄存器。只要计时器计数寄存器达到 0, 就会发生计时器中断。低分辨率时间等于调用该 API 时发生的时间中断总数。高分辨率时间等于, 计时器中断数与计时器周期寄存器的乘积, 与计时器计数寄存器的和。

如果剖析代码所需的时间比两个连续计时器中断间的时间周期长, 就会丢失那些被剖析的代码里隐蔽的计时器中断。丢失计时器中断相当于破坏低分辨率时间。当低分辨率时间被破坏时, 也自动破坏了高分辨率时间, 因为计时器中断的记录数是错误的。如果剖析代码需要花费很多周期, 就要确保降低配置文件里计时器的中断率, 防止丢失中断, 这样就可避免这个错误。如果应用程序使用计时器中断触发系统里的其他事件, 上述方法不是一个好的解决方案。这本质上就意味着, 如果 STS 对象屏蔽中断的时间比计时器中断率, 就不能使用 STS 对象来剖析代码。

另一种情况下, 即回零时, CLK\_gettime() 和 CLK\_gettime() 可能返回不正确的值。低分辨率和高分辨率时间表示为 32 位值, 可能会达到它们的最大值  $2^{32} - 1$ 。这些时间值达到最大值  $2^{32} - 1$  后变成 0。所以, 如果剖析代码所花费的时间比低/高分辨率时间达到各自  $2^{32} - 1$  所需的时间长, 就会发生回零, 返回的时间值就不再正确。在 C5000 上将计时器周期寄存器设置为 0xFFFF, C6000 设备设置为 0xFFFFFFFF, 就可以获得一个较好的时间分辨率。设置计时器周期寄存器里的这些值, 就能得到 DSP/BIOS 使用的 CLK\_gettime() 和 CLK\_gettime() 的优化版本。

### 5) 计时器 ISR 的开销

系统上的计时器 ISR 是有开销的。当计时器 ISR 服务时, 它调用 CLK\_F\_isr() 函数, 该

函数在硬件中断的上下文里执行。完成一些基本操作后，计时器 ISR 调用所有的时钟函数。一旦时钟函数已经执行了，通过 SWI 将控制转移给 PRD 模块。在 PRD 模块里，系统检查 PRD 函数的计数器是否达到 0，并为达到 0 的计数器运行相应的函数。所以，计时器中断事件里执行的处理数目依赖于配置的时钟函数的数目。随着时钟函数数目的增加，计时器中断服务所需的时间增长。所以，建议优化你的系统，使得系统里只有必需的 CLK 函数。不需要的时钟函数除了在计时器 ISR 里增加额外的开销外，别无益处。

详细的基准测试值请见相关参考文档。

## 3.2 NDK (Network Development Kit)

### 3.2.1 NDK 简介

TI 公司为了使其产品适应数字化网络的需求，推出了传输控制协议/因特网协议 (TCP/IP) 栈。此协议栈可使基于数字信号处理技术的因特网终端的网络连接成本降低 50% 以上。NDK (Network Development Kit) 是 TI 公司针对 TCP/IP 协议栈而推出的新型开发包。NDK 是对用户免费开放的，开发人员可以从官网处下载适用于自己芯片的不同版本的 NDK，以降低开发基于数字信号处理器的网络应用的时间和成本。但是在开发者将使用 NDK 开发的产品应用于市场的时候，需要向 TI 上交相应的费用。

NDK 可用于测试 TI 的 TCP/IP 协议栈的功能和性能，以满足各种不同应用对网络连接的需要。制造商可以利用 TCP/IP NDK 迅速在 DSP 应用上集成协议栈，从而在完成目标硬件之前就可以开始系统软件部分的设计。此外，TCP/IP NDK 还带有以太网子卡，配备了媒体访问控制层 (MAC) / 物理层 (PHY)，从而省去了网络处理器及相关软件，使整体单位成本下降 50% 以上。

NDK 主要的组件包括：(1) 支持 TCP/IP 协议栈程序库，其中主要包含的库有支持 TCP/IP 网络工具的库、支持 TCP/IP 协议栈与 DSP/BIOS 平台的库、网络控制以及线程调度的库 (包括协议栈的初始化以及网络相关任务的调度)；(2) 示范程序，其中主要包括 DHCP/Telnet 客户端、HTTP/数据服务器示范等；(3) 支持文档包括用户手册、程序员手册和平台适应手册。

NDK 是一个软件层 (实际上是许多层的集合)，它用到了实时内核 DSP/BIOS，可以用其来配置、控制并且调用软件以太网媒体访问控制寄存器 (EMAC, Ethernet Media Access Controller) 外设。有两种类型的 EMAC：(1) 用户知道 sockets (套接字) 程序，仅仅简单的让 NDK 作为 EMAC；(2) 用户在 EMAC 上层运行自己的软件，他需要知道 EMAC 结构的细节。

### 3.2.2 NDK 的基本架构和 API 函数

NDK 的设计目的是要提供一个完整的 TCP/IP 功能环境。NDK 通过编程接口与本地操作系统 (DSP/BIOS) 和底层硬件相互隔离，本地操作系统 (DSP/BIOS) 被抽象成一个操作系统适应层 (OS Adaptation Layer)，底层硬件被抽象成一个硬件抽象层 (Hardware Abstraction Layer)，两个抽象层的函数库分别为 os.lib 和 hal.lib，它们是 NDK 与本地操作系统和底层硬

件的接口。TCP/IP 协议栈（NDK）的结构如图 3.21 所示。其中，stack 库是主要的 TCP/IP 网络功能库，它包括了从底层链路层到上层套接口层的所有功能。该库是基于 DSP/BIOS 操作系统的。

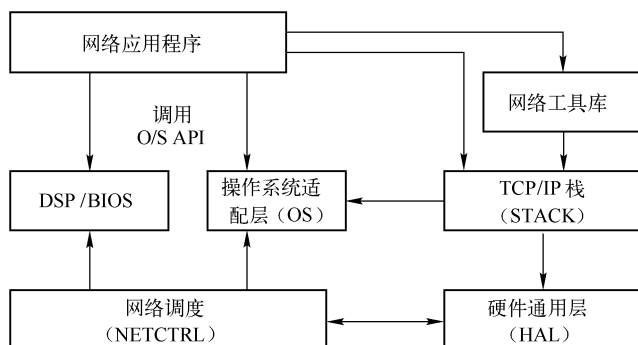


图 3.21 TCP/IP 协议栈组织结构

netctrl 库或叫网络控制库从某种意义上来说是整个协议栈的核心。它控制着 TCP/IP 和外界的联系和互动，在所有的模块里面，它对于 TCP/IP 协议的运行是最重要的。

其职能包括：

- 1) 初始化 TCP/IP 协议栈和底层驱动；
- 2) 启动并保持系统的配置；
- 3) 与底层驱动接口并安排底层事件传递进入 TCP/IP 协议栈；
- 4) 退出时将系统配置导出并清理底层驱动。

下面以 NDK1.94.1 为例，来说明 NDK 开发包的基本内容。

首先，在 TI 的官网下载 NDK1.94.1，进入官网直接在里面搜索即可。下载后直接点击安装即可，一切路径最好按照默认路径安装。在安装前要确保所用 PC 已经安装好了 CCS 软件（本文使用的是 CCS3.3 版本），并且 CCS 也是按照默认路径安装（这里要求使用默认路径安装的原因是 TI 许多库的搜索路径都是默认的，如果不这样安装的话会经常发生文件搜索不到或者需要再配置路径的问题）。

安装完成后，在 C 盘的 CCStudio\_v3.3 文件夹中会出现一个新的名为 ndk\_1\_94\_1 的文件夹，文件夹是 NDK 开发包的内容。此文件夹中最开始只存在一个文件夹，即 packages 文件夹，继续点击 packages\ti\ndk 里面会出现许多文件及文件夹，如图 3.22 所示。



图 3.22 路径 C:\CCStudio\_v3.3\ndk\_1\_94\_1\packages\ti\ndk 中的文件

其中主要的文件为：lib 库文件夹，里面存放上文提到的几个库文件；example 为例程文件夹，包含一些例程工程及文件。在 example 中只有一些 C 文件而不存在什么工程文件，是因为 NDK1.94.1 安装包里面只存在一些基本的库文件、源文件以及头文件，具体的工程仍需要在 TI 的官网下载对应版本的 NDK 补充文件。本书以 DM642 为例，下载补充文件并安装。安装好后，在 C:\CCStudio\_v3.3\ndk\_1\_94\_1 路径下出现了 dm642 文件夹，文件夹中包含有 DM642 所用到的 NDK 例程。设置好 Setup CCStudio v3.3 后，打开 CCS3.3 软件，在 File 下拉菜单下，选择 Open 选项，在弹出的框图中选择路径 C:\CCStudio\_v3.3\ndk\_1\_94\_1\dm642\ti\ndk\example\serial\client\evmdm642 中的 client.pjt 工程文件，这样就打开了一个基本的 NDK 例程。第一次打开这个例程，默认的是把所有最复杂的网络协议、功能都加上结构。开发人员往往用不到这么多复杂的东西，这就需要开发者弄清楚每个文件都实现什么功能，之后根据工程需要来有的放矢地选择不同的文件来进行开发。

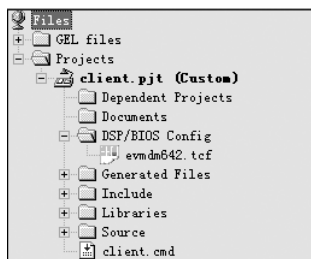


图 3.23 DSP/BIOS 配置文件位置

DSP/BIOS 的使用。

点开 DSP/BIOS Config 左边的加号，显示其目录下的 tcf 格式的文件，如图 3.23 所示。双击 tcf 文件，将打开整个配置文件，如图 3.24 所示。这个是 DSP/BIOS 配置文件。DSP/BIOS 为 DSP 处理器内的实时操作系统，在内部可以配置线程、硬件中断、软件中断、任务等，具体功能详见 DSP/BIOS 的有关章节。NDK 开发包用到了 DSP/BIOS 来配置整个系统的内存分配、建立静态任务、设置周期函数等。所以，要了解 NDK 的开发过程必须先掌握



图 3.24 DSP/BIOS 配置文件的内容

### 3.2.2.1 DSP/BIOS 文件中的关键配置

#### (1) Global Settings

点开 System 左侧的加号，会显示其菜单下的许多配置。其中一条为 Global Settings，点

击右键选择 Properties 选项，会弹出其参数配置表，如图 3.25 所示。

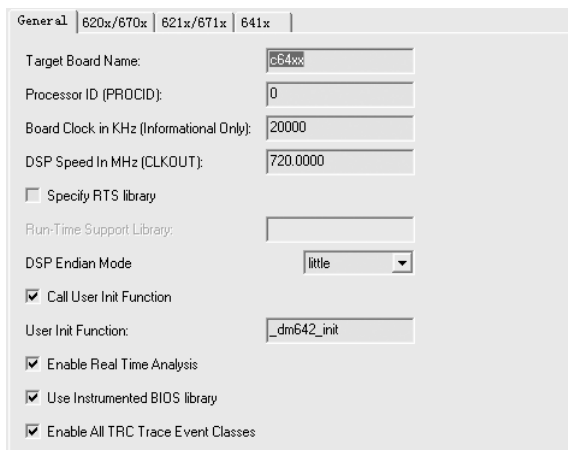


图 3.25 Global Settings 的 Properties 选项内容

其中，板子时钟（Board Clock）与处理器速度（DSP Speed）需要自己根据所选的开发板与芯片的不同而改变。DSP 端模式（Endian Mode）需要选择 little 模式。User Init Function 为初始化函数，一般为基础的配置文件，TI 公司已经写好，开发者尽量不要修改。\_dm642\_int 前面的下划线“\_”不能省略，因为 dm642\_int 在整个工程中是一个 C 语言编写的函数（在 dm642int.c 源文件中），而在 DSP/BIOS 上面书写的函数应该为汇编模式，所以每个函数前都要加下划线。其他的 3 个选项都使用默认配置即可。因为使用 C64 系列的处理器，所以在 641x 选项卡下会有一些缓存 L2、EMIF 地址配置等，都使用默认配置即可。

## （2）钩子函数（HOOK）

打开 HOOK 选项菜单，里面会显示此工程配置的钩子函数，如图 3.26 所示。第一个为默认的 HOOK 函数，而第二个函数 hookNdk 为配置的函数，里面设置了初始化函数以及创建函数，此部分使用默认配置即可，不需要开发人员改变。

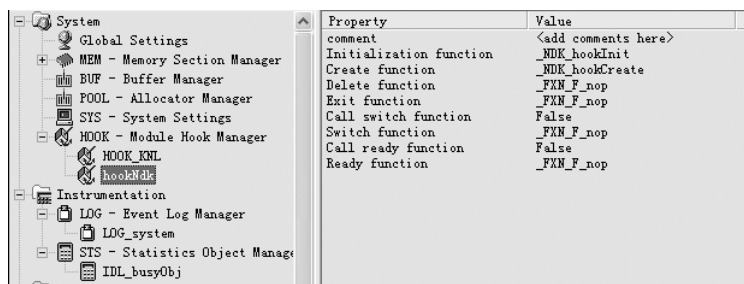


图 3.26 HOOK 函数

## （3）周期函数 PRD

如图 3.27 所示，打开 Scheduling 选项卡下的文件，其中 CLK 为周期时钟，这里面的 PRD\_clock 为默认的配置。开发人员不需要改变。

PRD 为周期函数选项，prdNdk 为配置的周期函数，周期为 100ticks，函数名称为 llTime-rTick。此函数会用到 DM642 内部的定时器，需要在开发过程中用到定时器的开发人员特别需要注意，不要和这个定时器冲突。



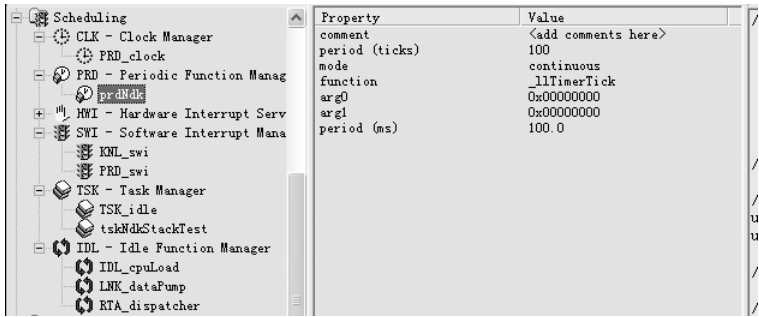


图 3.27 Scheduling 下的配置文件

(4) 软件中断 SWI 与硬件中断 HWI

首先点开 HWI 界面，会出现如图 3.28 所示的结构，这里面类似于配置汇编的中断向量表，只是把配置变得更加形象化。具体内容详见 DSP/BIOS 章节内的 HWI 部分。在 NDK 开发包中，我们没有用到硬件中断，这里只需用默认配置即可，在开发过程中开发人员可以根据自己的需要来设置硬件中断，但是要记住上文提到过的定时器。

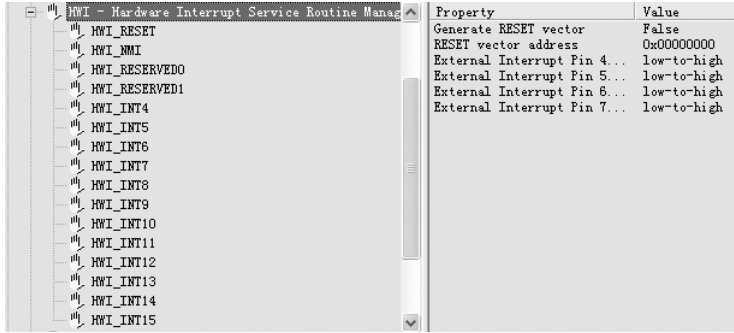


图 3.28 HWI 配置

再点开 SWI 左侧的加号，从图 3.27 中就可以看出，里面存在 2 个 SWI 配置。KNL\_swi 为默认的软中断函数。PRD\_swi 为 NDK 配置软件中断，调用的中断函数为 PRD\_F\_swi，优先级设为“1”。

(5) TSK 任务管理

点开 TSK - Task manager 左侧的加号，可以看出里面存在 2 个已经配置好的静态任务，如图 3.29 所示。从图中可以看出，第一个 TSK\_idle 为任务的空闲线程，此为默认配置。而

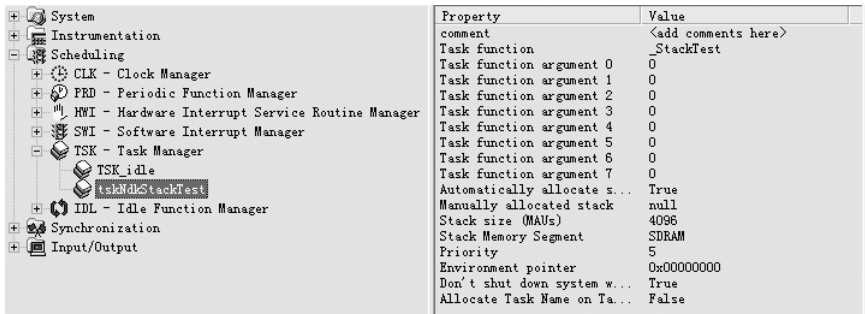


图 3.29 TSK 任务配置



第二个名为“tskNdkStackTest”的静态任务是 NDK 配置的网络调度任务，在工程执行完主函数后会进入这个任务中去。由图可知，此任务的优先级为 5，高于后台线程任务，并且没有其他更高优先级的任务存在，所以此任务会优先执行。任务的函数名为 StackTest，是执行任务时所调用的函数，配置的栈的大小为 4 096，存储位置为 SDRAM。

后台线程 IDL、同步、输入输出等模块都使用默认配置即可。至此，DSP/BIOS 的配置部分就介绍完毕。这里建议开发人员尽量不要修改 NDK 使用到的配置，否则很容易造成未知错误。

### 3.2.2.2 NDK 中的库文件

点开 Libraries 左侧的加号，会出现如图 3.30 所示的一些 lib 文件即库文件。它们为 TI 公司自己编写并优化过的一些功能函数，是整个 NDK 的基础所在。它们组成了 NDK 中的 TCP/IP 协议栈，逻辑结构可以参见图 3.21。lib 文件不能在 CCS 中双击打开，只能通过文本编辑器单独打开。有些库文件在用文本编译器打开后会

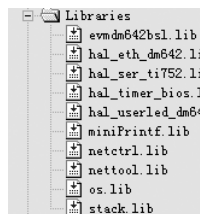


图 3.30 NDK 中的库文件

发现其内容是标准的 C 语言程序，而有些则是 0、1 的“乱码”。这是因为部分库函数 TI 提供了源码，直接封装在了 lib 文件中，而有些库函数则没有提供源码，至于那些所见到的乱码其实是编译后相对应的二进制文件。一般的“文件名.lib”是小段模式，而“文件名.e.lib”为大端模式，“文件名\_jumbo.lib”表示该库对巨帧的支持。当然，对于不同版本的 NDK，库函数的名字也会不同，但是一些关键字的名字还是一样的。比如有的版本为 hal\_ser\_ti752.lib，而有的版本为 hal\_ser\_stub.lib。

针对不同开发板（HAL，hardware abstraction layer）库文件是不相同，它们是涉及硬件配置的库文件，而其他库文件可以通用。NDK 的 HAL 库文件是根据平台和设备来制订的。NDK 的安装包都包含了针对所有平台的 NDK 支持包。不同平台或设备的特定的 HAL 库文件都存放在 <lib/hal> 目录之下。

hal 目录下面是一系列的驱动文件，是硬件外设与 NDK 之间的接口，包括：定时器、LED 指示器、以太网设备、串口。这些驱动文件一般都是用“hal\_类型\_设备.lib”来命名，如 hal\_eth\_dm642.lib 是基于 DM642 的以太网驱动。有些文件也不是按照上面的命名方式，但是其类型或者功能仍然能够从命名中有所了解。

下面是一些库文件的介绍：

- “hal\_eth\_stub.lib”，Ethernet Stub Drivery，以太网口驱动；
- “hal\_ser\_stub.lib”，serial Stub driver 串口驱动；
- “hal\_timer\_bios.lib”，Timer driver using DSP/BIOS PRD object 基于 DSP/BIOS 周期对象的定时器驱动；
- “hal\_userled\_dm642.lib”，LED 口的驱动程序；
- “nettool.lib”，网络库函数库文件，包含所有基于 sockets 的网络服务和一些用于网络应用开发的附加工具；
- “os.lib”，系统适应层库文件，将系统函数调用映射到 DSP/BIOS 函数调用；
- “miniprintf.lib”，占用资源很小的打印输出库文件；
- “stack.lib”，NDK 库文件，主要的 TCP/IP 网络协议栈，包含了上到 socket 层下到以太网/PPP 层的所有功能。

- “netctrl.lib”，网络初始化、控制库文件，是堆栈的管理者，控制 TCP/IP 协议和外界的交互作用。

在一个网络程序执行之前，TCP/IP 协议栈必须正确配置并初始化。为了方便配置和初始化，NDK 提供了网络控制模块（NETCTRL.LIB）的源程序。该模块是协议栈的配置、初始化和事件调度的核心。对于该模块的深入理解，是编写一个完备的网络程序的关键。

几乎所有网络控制活动都发生在“NETCTRL”任务线程（也称为“调度线程”）中。该线程由程序员通过 DSP/BIOS 配置工具或者通过 DSP/BIOS 的 API 函数来创建。在协议栈所有的示例程序中，有一个通过 DSP/BIOS 配置工具创建的主线程。该主线程是程序的入口，并最终演变成“调度线程”，直到程序结束该线程才返回。网络任务如 FTP、HTTP 等，并不在该主线程中执行，而是另外开辟新线程。

要想调用 NETCTRL API 函数，必须创建系统配置。该配置是一个句柄对象，包括系统的许多参数，由这些参数控制着协议栈的运行。典型的配置参数包括：

- 1) 网络主机名；
- 2) IP 地址和子网掩码；
- 3) 默认路由器 IP 地址；
- 4) 执行的服务（如 HTTP、FTP、DNS 等）；
- 5) 域名服务器的 IP 地址；
- 6) 协议栈的参数（IP 路由、协议栈缓冲区大小等）。

这些库都是 NDK 开发过程中所需要的极为重要的文件，开发人员在开发过程中尽量保持原有的这些库函数，以免造成未知错误。只有当开发板内存不够时才考虑精简库函数。

### 3.2.2.3 NDK 中的源文件

点开 Source 左侧的加号，会出现如图 3.31 所示的文件，这是 NDK 开发包中所有的源文件。每个文件都是实现部分协议的功能，具体可以根据其文件名大致读出其含义。

dm642init.c 为开发板初始化函数，是 DSP/BIOS 所调用的，在上文中已经提到。这个源函数是每个工程中必须存在的。

evmdm642\_osd.c 也是默认的配置文件，包括 CHIP 号、CPLD 配置、硬盘配置等，此文件最好能够保留。

client.c 是整个工程的主体函数，里面包含了整个 NDK 的主要架构。打开 client.c 文件，会看到里面有一个主函数 `int main() {}`。但是其内部没有任何语句，这是因为这个 NDK 工程都是通过调度线程来执行的，在工程执行完主函数后，会自动跳到所建立的 Task 任务中去，在上一节的内容中有这方面的介绍。任务调用的函数为 `StackTest()`，此函数为这个 NDK 的主要结构。

(1) `StackTest()` 函数。

(2) `NC_SystemOpen`：首先利用 `NC_SystemOpen` 打开一个系统，这个是所有工程的第一步。当打开成功的时候，返回值为 0，如果不为 0，则失败，进入死循环。

之后，为 NDK 的一系列网络配置，针对不同的工程、不同的协议标准，这些配置会存在不同，具体操作需要参照 TI 公司提供的技术手册来进行修改。

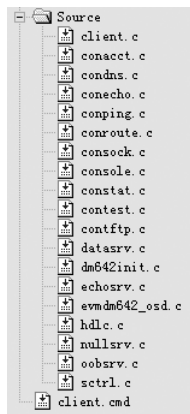


图 3.31 NDK 文件中的全部 source 文件

正确的配置被导入以后，网络（和网络调度线程）通过一个 NETCTRL 函数 NC\_NetStart() 启动，系统没有关闭之前，NC\_NetStart() 函数是不返回的。它的返回值是一个关闭代码。通常该系统的关闭方式会决定协议栈是否重新启动，比如改变配置后需要自动重启，一般有以下代码：

```
do
{
    rc = NC_NetStart( hcfg, NetworkOpen,
        NetworkClose, NetworkIPAddr );
} while( rc > 1 );
```

该例子中，如果 NC\_NetStart() 返回值大于 0，就重新启动网络。

NC\_NetStart 函数里面有 4 个参数，hcfg 为配置参数产生的函数句柄；NetworkOpen 为动态创建网络任务的函数，里面调用了 TaskCreate API 函数来动态创建任务线程。不同的服务器端的协议会创建不同的任务。

NetworkClose() 函数为关闭任务的函数。先是 sctrlAbort、fdCloseSession（句柄）函数关闭串口，ConsoleClose 关闭活动控制台，TaskDestroy（句柄）删除任务。

NetworkIPAddr() 此函数为配置 IP 的函数，可以管理加入一个新的 IP 或者删除一个已有 IP。

其他源文件都是一些协议，根据需求来自己选择。

静态任务就是 NDK 的基本架构，而具体的、不同的协议内容，则是通过创建动态任务来完成的。在开发中，根据需要来选择适合的协议、适合的源文件、不同的动态任务，还要根据需要进行修改。

### 3.2.3 NDK 应用实例

本小节以最小的一个工程为例，讲解 NDK 的基本流程。本例选择最简单的 echo 协议，此协议为测试协议，是一个 ACK 流程。工程只用 UDP 这种无连接的模式，而省掉了 TCP 的方式。

根据 TI 提供的实验手册进行修改，详见《Getting Started With the C6000 Network Development Kit》。

在静态任务中，需要先修改部分配置。

- (1) CI\_SERVICE\_TELNET telnet；此声明为 telnet，可以删掉。
- (2) 在下面的配置文件中，把 telnet 等的配置全部删掉，换成如下配置：

```
if( inet_addr( LocalIPAddr ) )
{
    CI_IPNET NA;
    CI_ROUTE RT;
    IPN      IPTmp;
    bzero( &NA, sizeof( NA ) );
    NA. IPAddr  = inet_addr( LocalIPAddr );
```

```

    NA. IPMask    = inet_addr( LocalIPMask );
    strcpy( NA. Domain, DomainName );
    NA. NetType = 0;
    CfgAddEntry( hCfg, CFGTAG_IPNET, 1, 0, sizeof( CI_IPNET ), ( UINT8 * ) &NA, 0 );
    bzero( &RT, sizeof( RT ) );
    RT. IPDestAddr = 0;
    RT. IPDestMask = 0;
    RT. IPGateAddr = inet_addr( GatewayIP );
    CfgAddEntry( hCfg, CFGTAG_ROUTE, 0, 0, sizeof( CI_ROUTE ), ( UINT8 * ) &RT, 0 );
    IPTmp = inet_addr( DNSServer );
    if( IPTmp )
        CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_DOMAINNAMESERVER,
                      0, sizeof( IPTmp ), ( UINT8 * ) &IPTmp, 0 );
}
else {
    CI_SERVICE_DHCPC dhcpc;
    bzero( &dhcpc, sizeof( dhcpc ) );
    dhcpc. cisargs. Mode    = CIS_FLG_IFIDXVALID;
    dhcpc. cisargs. IfIdx    = 1;
    dhcpc. cisargs. pCbSrv = &ServiceReport;
    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DHCPCLIENT, 0,
                sizeof( dhcpc ), ( UINT8 * ) &dhcpc, 0 );
}

```

### (3) 在 rc 后面加入新的配置

```

rc = 8704;
CfgAddEntry( hCfg, CFGTAG_IP, CFGITEM_IP_SOCKUDPRXLIMIT,
             CFG_ADDMODE_UNIQUE, sizeof( uint ), ( UINT8 * ) &rc, 0 );

```

(4) 在 NetworkOpen 函数与 NetworkClose 函数中，删除其他的动态任务，只保留一个。这里新建一个源文件，命名为 udp\_send. c，而调用的动态任务函数仍然命名为 udp\_send。

(5) 在 udp\_send. c 文件中新建函数 udp\_send()，此函数是新的动态任务函数。

```

SOCKET    sudp = INVALID_SOCKET ;           //声明套接字
    struct    sockaddr_in sin1;
struct    timeval timeout;                  //定时器配置,超时会执行相应操作
int        tmp;
    HANDLE    hBuffer;                      //缓冲区句柄
    unsigned char    * pBuf = 0;           //缓冲区
    int LocalPort = 88;                      //端口号
    //char * DestIPAddr = "192. 168. 1. 91" ;
    fdOpenSession( TaskSelf( ) );           //打开任务
    bzero( &sin1, sizeof( struct sockaddr_in ) );

```

```

sin1.sin_family = AF_INET;
sin1.sin_len    = sizeof(sin1);
sin1.sin_port   = htons(LocalPort);
sudp = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);    //建立 UDP 的 SOCKET
if(sudp == INVALID_SOCKET)
{
    printf("Failed Socket Create (%d) \n", fdError());
    goto leave;
}

if(bind(sudp, (PSA) &sin1, sizeof(sin1)) < 0)
    goto leave;

timeout.tv_sec    = 0;    //设定超时时间,第一个为秒,第二个为微秒
timeout.tv_usec   = 100000;
printf("Srv Initialized \n");

```

#### (6) for 循环

进入 UDP 的主体部分,使网络一直存在下去,本例用了一个死循环,整个 UDP 流程一直在这个循环中执行下去。

```

for(;;)
{
    fd_set ibits, obits, xbits;
    int cnt, cnt2;
    FD_ZERO(&ibits);
    FD_ZERO(&obits);
    FD_ZERO(&xbits);
    FD_SET(sudp, &ibits);
    if(FD_ISSET(sudp, &ibits))
    {
        tmp = sizeof(sin1)
        .....
    }
    .....
}

```

#### (7) 接收和发送

UDP 协议中接收和发送主要用到 2 个函数,一个是 `recvncfrom`, 另一个是 `sendto`。  
`recvncfrom` 的格式如下:

```
cnt = (int)recvncfrom(sudp, (void **) &pBuf, 0, (struct sockaddr *) &sin1, &tmp, &hBuffer);
```

在调用完此函数后,需要释放缓冲区,调用 `recvncfree(hBuffer)`。

`sendto` 的格式如下:

```
sendto(sudp, pConfirm, 4, 0, (struct sockaddr *) &sin1, sizeof(sin1)); //4 为发送 char 型的个数
```

### 3.3 DDK (Device Driver Kit)

DSP/BIOS Driver Developer's Kit (DDK) 是 TI 为简化驱动程序开发的, 为 TMS320 系列 DSP 及其 EVM 板等提供的驱动程序开发套件。该套件为 TMS320 系列各种外围器件提供完整的标准化驱动程序模型, 使得驱动程序可以很方便地移植到其他应用中, 大大提高了驱动程序开发的效率。DDK 是对每种 TMS320 系列 DSP 都提供的芯片支持库 (Chip Support Library - CSL) 的补充, CSL 提供对外围器件寄存器配置及初始化等的低级控制, DDK 完全通过 CSL 来对外围器件进行控制。简单来讲, DDK 建立在 CSL 上层, 所以用 DDK 来开发驱动程序将更为快捷且可移植性更好。

DDK 为开发驱动程序定义了标准模型和一系列的 API。为简化程序设计, 标准模型又被分为两个层次, 其中高层称为 class driver, 低层称为 mini-driver。class driver 与器件相对独立, 完成诸如缓冲区管理和请求同步等功能, 同时扮演着与 API 和 mini-driver 二者接口的角色。mini-driver 完成特定的器件初始化和控制功能, 它符合 IOM (I/O Mini-driver) 的接口标准。DDK 的这种分层结构使得驱动开发人员仅需了解单一的 mini-driver API 就可以完成整体外围器件的驱动设计, 而且这一过程比设计整个驱动程序要简单得多, 因为 class driver 控制了缓冲区管理和同步等。DDK 提供 3 种 class driver: 分别为 SIO/DIO、PIP/PIO 和 GIO, 它们都可以和任何 mini-driver 结合使用。

#### 3.3.1 DDK 概述

DDK 提供了: 对 TMS320 DSP 外设的全功能的设备驱动; 一个具有证明文件的驱动模型, 对驱动开发的方法进行了标准化; 一系列可重复利用的驱动模块, 排除了开发代码的重复性。分层的驱动模式同时给应用程序集成商提供了许多益处。标准的 mini-driver API 函数允许 class drivers 协同 mini-driver 工作。编解码器的 mini-driver 能够运用在任何的预定义的 DSP/BIOS I/O 模式, 例如 SIO 或者 PIP。因为需要一个对应的不同的 API 函数, 所以没有必要重新写一个驱动。如果开发者想实现一系列可供选择的 I/O API 函数, 他可以实现一个新的 class driver 或者扩展一个已经存在的 class driver。

class driver 提供了 3 种可重复利用的模式。

SIO/DIO: DIO 适配器允许 DSP/BIOS SIO 模式同 mini-driver 共同使用, 两种模式的组合是一个 class driver, 应用程序调用内部使用 DIO 适配器的 SIO 函数, 而不是 DIO API 函数。

PIP/PIO: PIO 适配器允许 DSP/BIOS PIP 模式同 mini-driver 共同使用, 它们的组合是 class driver, 当使用 class driver 时, 应用程序将调用 PIP 和 PIO 的 API 函数。

GIO: GIO 模式实现一系列的 I/O 的 API 函数, 这些 API 函数可能被当作 DSP/BIOS 到 mini-driver 的接口, GIO 模式是其 class driver 选项。GIO class driver 被设计用来支持 I/O API 函数简单的扩展来满足专业设备的需求。例如: 对异步数据处理的扩展设计——ASYNC 模式。此外, 专业的 API 函数可以很简单地建立, 如成帧视频的 API 函数。

在 class driver 之外, TI 公司还提供一个 mini-driver 的选项, 可以清楚地阐述怎样开发针对不同外设的特定设备代码。现有的 mini-driver 可以作为一个初始模块来为新的外设提供开发支持。



3.3.2 DDK 的基本结构

3.3.2.1 两层设备驱动模式

随着 DSP 实时系统变得更加复杂、新技术的不断出现，外设的品种和数量不断增长。编写和移植这些外设的驱动已经成为依赖于硬件和 operating 系统的任务。例如内存占用、响应时间、电源管理等约束条件对 DSP 系统是一个比较大的困难。

设备驱动的开发已经从一个基于已有模型的开发模式中受益，此模式依据功能将外设区分为硬件独立与软件相关两个层。每个层使用通用接口，这样相似外设的驱动软件的主要部分可以重用，最终简化了驱动开发过程。

**Class driver:** 这类驱动通常提供多线程 I/O 请求的序列化和同步，并处理设备实例管理。在典型的实时系统中，只有少数这类驱动程序，作为这类设备应用程序的示范，包括 I/O 块、I/O 字符和视频。

**Mini-driver:** 这类驱动利用具体设备的 mini-driver 来操作一个特定的设备，方便了应用程序软件。

1. 应用程序架构概述

图 3.32 描述了两层设备驱动模型中各层之间的关系。高层的应用程序和 mini-driver 相互没有直接的关联。它使用一个或多个 class driver 与 mini-driver 接口。每个 class driver 对应用程序提供了一个 API 函数，并与 IOM mini-driver 接口进行通信。一个 class driver 使用 DSP/BIOS API 为操作系统服务。它调用标准 mini-driver 接口来使用硬件外设。DSP/BIOS 当前定义了图 3.32 所示的 3 个 class driver: PIP/PIO、SIO/DIO 和 GIO。对于 PIP/PIO 和 SIO/DIO class driver，这些应用程序利用的 API 函数存在于 DSP/BIOS PIP 的 SIO 函数内。这些 API 函数关联相应连接 mini-driver 的适配器。对于 GIO class driver，应用程序调用那些直接连接 mini-driver 的 API 函数。在一个应用程序中可能同时存在不止一种类型的 class driver。应用程序开发者能够选择其中的一个或者全部来为系统服务，而 mini-driver 开发者不需要编写 class driver。

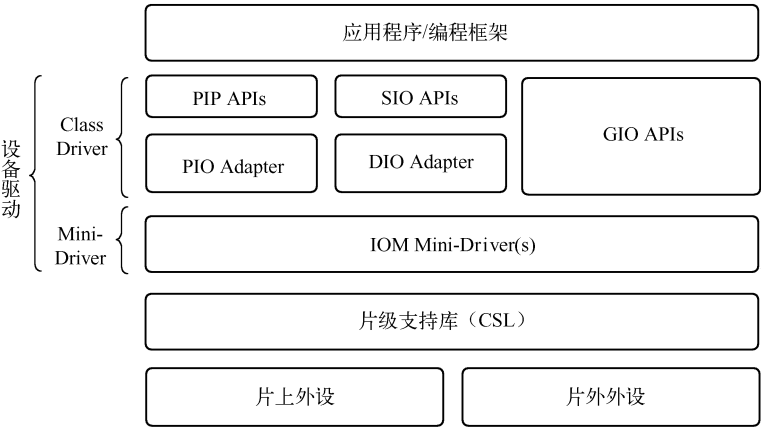


图 3.32 显示设备驱动组成的应用程序架构

每一个 mini - driver 带有标准的 mini - driver 接口函数，这些函数是 class driver 用来访问硬件和 DSP/BIOS 设备驱动管理。Mini - driver 用片上支持库（CSL）来连接外围硬件寄存器、内存和中断源。一些 mini - driver 可以随意地包含特定编解码器的子驱动。

## 2. 驱动初始化和装载

每一个应用程序调用的 DSP/BIOS 模块包含一个在 DSP/BIOS 初始化时所调用的初始化程序。Mini - driver 类似于其他 DSP/BIOS 模块，每一个注册的 mini - driver 的初始化函数在 DSP/BIOS DEV 模块初始化时被调用。Mini - driver 初始化函数的调用顺序是由 mini - driver 配置的。每一个 mini - driver 函数表输出一个在驱动初始化函数之后被 DSP/BIOS 调用的绑定的函数（mdBindDev）。

## 3. 设备和信道实例

设备驱动模型包括两种类型的对象实例。

### 1) 设备实例

这是一种实际外围设备的抽象，如音频编解码器或外部视频端口。在 DSP/BIOS 的设备表中注册（配置）设备实例。应用程序是使用它们的“逻辑”名称。如果要配置多个设备实例，每个实例在 DSP/BIOS 设备表中都有一个唯一的逻辑名称。

### 2) 通道实例

这是一个应用程序和设备实例之间通信路径的抽象。通过调用 mini - driver 的 mdCreateChan 功能来建立通道实例，其运行的结果是调用这一类驱动程序（如：SIO\_create、PIO\_create 或者 GIO\_create）。

对物理设备通过设备例程抽象后，在一个 DSP 系统中可能不止一种类型。使用相同的 mini - driver 代码用于多个设备保持不同的驱动状态，这是驱动程序的一个重要特征。DSP/BIOS 驱动程序不需要支持多实例，但是如果多个硬件外设具有相同的类型，仍然强烈建议支持多实例。例如许多 TI 的 DSP 有多个 McBSP 外设，所以在 DDK 驱动程序中都支持使用 McBSP。

一个给定的设备实例可以支持多通道实例。操作定向模式是通道实例的一个重要属性。对于 mini - drivers，通道模式是数据输入、数据输出或者数据输入和输出（双相）。如果一个应用程序试图使用不支持的模式建立通道，mini - drivers 不支持一个或多个模式返回一个错误状态。

### 3.3.2.2 驱动数据流

图 3.33 表明 class driver 在使用 mini - driver 时会调用流模式。IOM 包（IOM\_Packets）是相应 mini - driver 请求的标准的数据结构，其包含了一个指向数据缓冲区的指针。

在应用程序和设备通信之前，mini - driver 的 mdCreateChan 函数必须返回给应用程序一个信道实例句柄。根据 class driver，所指出的通道实例代表点到点的数据流（SIO）或者管道（PIP）。在任何情况下，通道句柄代表一个独特的应用程序和 mini - driver 之间的通信路径。所有跟驱动有关的后续操作都使用这个通道句柄。

每个通道所占用的数量（例如内存）取决于 mini - driver 和一个通道适配器的实现。一个通道对象通常保持基于通道模式的数据域、I/O 请求队列和驱动状态信息。总内存大小可能针对不同的适配器和 mini - driver，应用程序应该在通道实例不再需要时删除它们以释放通道资源。

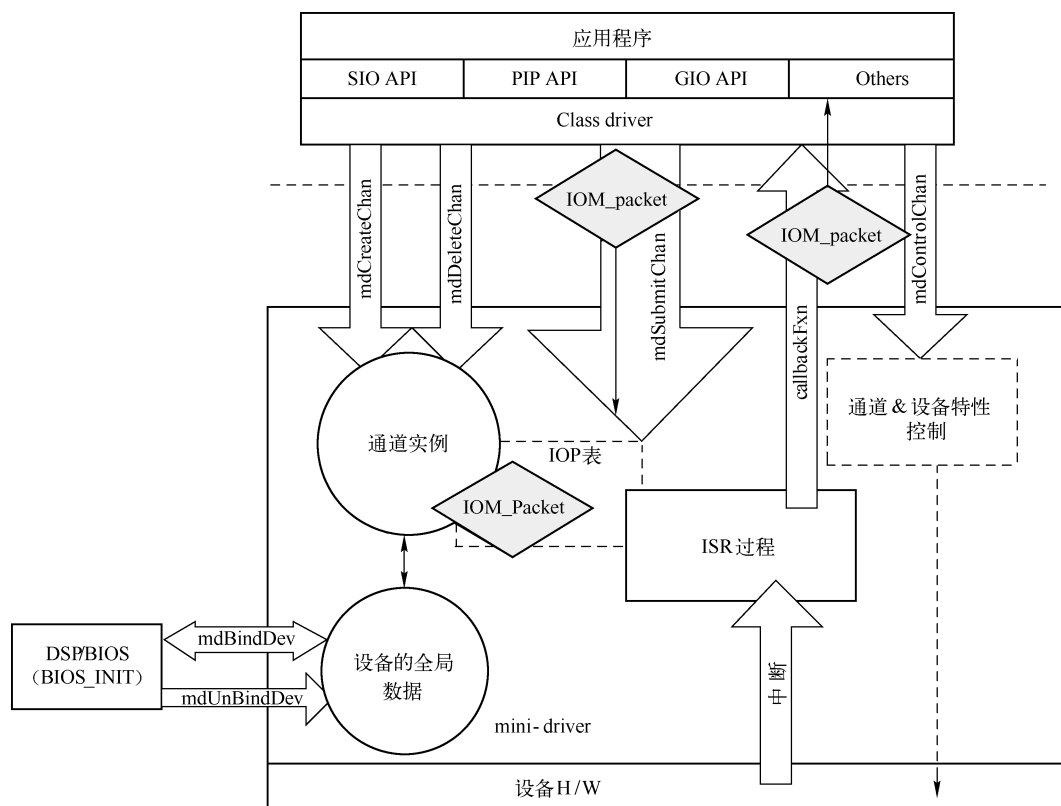


图 3.33 mini-driver 操作流程

IOM 驱动利用 IOM\_Packet 数据结构来向 mini-driver 提交请求，这个标准数据结构是等效的 DSP/BIOS DEV\_Frame 结构，这种结构已经改为包含命令（cmd）字段和一个命令（cmd）状态字段。IOM\_Packet 结构定义如下：

```
typedef struct IOM_Packet {
    QUE_Elem link;
    Ptr addr;
    Uns size;
    Arg misc;
    Arg arg;
    /* two fields added for use by IOM */
    Uns cmd;
    Int status;
} IOM_Packet;
```

这两个新字段扩展了 DSP/BIOS DEV\_Frame 结构，使一组更丰富的 I/O 操作在应用程序和 mini-driver 之间传递。应用程序自身没有 IOM\_Packet 结构，仅 class driver、适配器和 mini-driver 使用这些包。mini-driver 命令（cmd）值常数允许 mdSubmitChan 函数被读、写、中断或者刷新通道等操作调用。

IOM\_Packet 结构是由 class driver 创建的并被 mini-driver 的通道对象引用。GIO/IOM 模

式调用的 IOM\_Packet 是一种固定大小的结构。一个 IOM\_Packet 表示单独一个到 mini - driver 的 I/O 请求。mini - driver 处理请求，通过调用 class driver 的 callback 函数来返回给 class driver 包。

当一个应用程序申请一个 I/O 请求时，class driver 在提交到 mini - driver 之前将其填充到 IOM\_Packet。下面表明了 GIO\_submit(GIO\_read、GIO\_write) 作用在 IOM\_Packet 中。

```
QUE_Elem    link;           /*    queue    link    */
```

内部队列调用，可能被 mini - driver 调用。

```
Ptr    addr;           /*    buffer    address    */
```

Class driver 对 bufp 参数进行设置，这个参数指向数据结构或者 buffer 数据。mini - driver 应该保护参数的值。

```
Uns size;    /*    buffer size    */
```

Class driver 用这个 size 来设置 buffer 数据结构的大小。如果实际的大小与请求的不同，则需要 mini - driver 刷新这个 size 所在的内存区间。

```
Arg misc;    /*    miscellaneous    item    */
```

Class driver 为特定的应用程序 callback 函数设置此参数，mini - driver 不能修改这个参数。

```
Arg arg;    /*    command for mini - driver    */
```

GIO 不会使用这个参数，可以被其他的 class driver 使用。

```
Uns cmd;    /*    command for mini - driver    */
```

Mini - driver 用这个命令来决定在 mdSubmitChan 执行哪个操作。Mini - driver 不能修改这个内容。

```
Int status;    /*    status    of    command    */
```

Mini - driver 应该在 callback 函数之前设置这个状态。如果操作执行成功，status 会被传递给 IOM\_COMPLETED。如果失败，mini - driver 会将 status 设置到适当的 IOM 误码。Class driver 向应用程序返回这个 status。

### 3.3.2.3 Class driver

DSP/BIOS 支持两种数据传输类型：一种是流模式，也称作 SIO 模型；另一种是管道模式，也称作 PIP 模型。它们共同的特点是：

- (1) 要求一个管道或者流分别有一个单独的读线和一个单独的写线；
- (2) 传输管道内或者流内的缓冲区是通过复制指针而不是直接复制数据来完成的；

(3) 两种模式是用来管理块状的 I/O。

这些数据传输模式都能通过提供给 class driver 的适配器与 IOM mini-driver 交互。第一种适配器是 SIO 适配器 (DIO)，使用 SIO 模式。第二种适配器是 PIP 适配器 (PIO)，使用 PIP 模式。

DDK 介绍了第三种传输模型，主要是针对文件系统 I/O 和 UART 应用程序。这种模式是基于流的同步 I/O 模式，此模式提供更多通用的读写 API 函数到应用程序，并且通过 GIO class driver 变得更加有效。GIO class driver 是为直接适应 IOM 接口而编写的，因此有一个内建的 IOM 适配层。对于许多特殊的 I/O 应用程序或者文件系统应用领域，GIO 是一种较好的数据传输模型，它比 SIO 和 PIP 提供的同步块 I/O 模式更好。在使用一个模式而非另一种时常常会存在重叠。

### 3.3.3 DSP/BIOS 设备驱动

#### 3.3.3.1 注册 mini-driver

在 DSP/BIOS 应用程序中注册 IOM mini-driver，必须对应用程序进行配置，可以通过 DSP/BIOS 配置工具或者 DSP/BIOS Tconf 脚本来进行配置。

在 DSP/BIOS 配置工具中配置 mini-driver 要遵循如下步骤。

(1) 在 Input/Output 树下面的 Device Drivers 目录下，右键选中 User-Defined Devices 创建一个新的设备对象。

(2) 修改对象名称。这个名称在应用程序代码中定义了外设。例如，如果 mini-driver 驱动编码器，应用程序标记其为 “/codec”，那么可以命名为 “codec”。

(3) 右键点击所创建的对象，选择 Properties。

(4) 在 Properties 表中，设置参数来匹配 mini-driver 文件。“init function” 是初始化函数，应该用汇编形式 (加下划线)。“function table pointer” 填入驱动函数表的名字，这个表需要匹配上面创建的对象。“function table type”：如果你打算用像创建的 mini-driver，选择 IOM\_Fxns；原先的 Dxx 驱动则用 DEV\_Fxns 函数表。后面的 3 个 device 需要根据是否使用了某个设备来决定如何配置这 3 个参数。

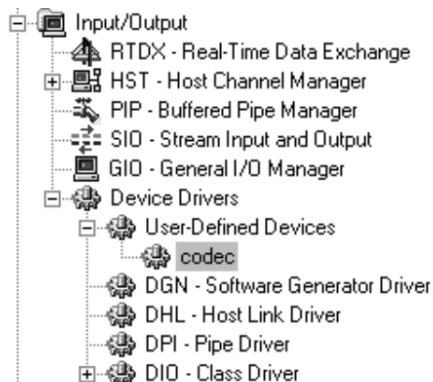


图 3.34 创建 codec

#### 3.3.3.2 配置 DIO class driver (针对 SIO)

如果应用程序使用 SIO (流 I/O) 模块的 DSP/BIOS API 函数，必须用 DIO 适配器来连接 mini-driver。SIO 模块和 DIO 适配器共同创建一个 class driver，SIO 函数同 DIO 适配器通信，DIO 适配器和 mini-driver 通信。根据是否选用 DIO 函数表的 callback，可以使用 TSK 为 DIO 函数配置，也可以用 SWI。如果使用 SWI，则需要选择这个表项。DIO 函数也可以同静态创建对象或者动态创建对象一起使用。DIO 对象只能在 IOM mini-driver 注册后才能创建。

例如，对 DIO 配置步骤如下。

- (1) 为 mini - driver 创建 UDEV 对象。
- (2) 设置 DIO - Class Driver manager：选中静态创建所有的 DIO 对象，如图 3.35 所示。仅仅在静态创建 SIO 流时，不选此项。
- (3) 右键点击 DIO - Class Driver，并选中 Insert DIO。
- (4) 重命名对象。
- (5) 右键选中所建对象，选则 Properties，设置如图 3.36 所示。
  - “ues callback version of DIO function”，当用 SWI 来使用 SIO 流时，不选此项；
  - “device name” 上面设定 UDEV 对象的名称；
  - “channel parameters” 设为 0x00000000（或者是一个指向传递给 mdCreateChan 的结构体的指针）。

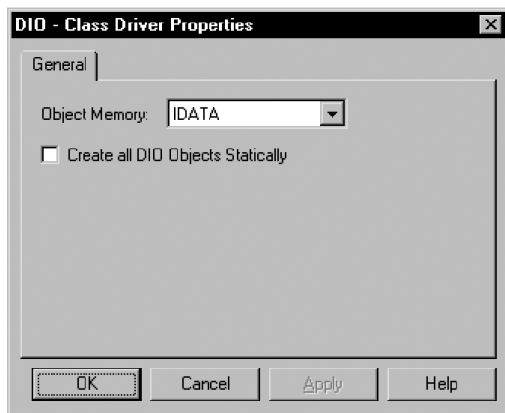


图 3.35 DIO - Class Driver 管理器

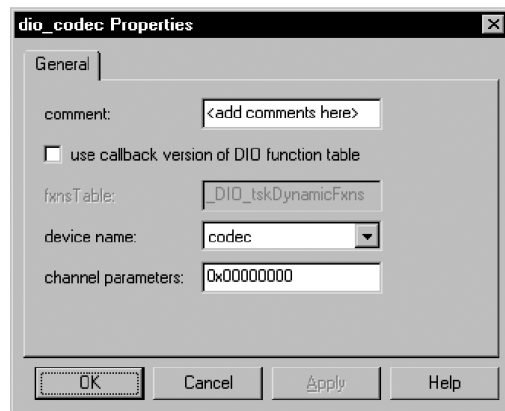


图 3.36 dio\_codec 配置

- (6) 创建 TSK 对象（也可能是用 SWI，但是不常见）。
- (7) 使用 SIO 组件 API 函数来创建和使用 SIO 流。
- (8) 使用适当的 mini - driver 库来连接应用程序。

### 3.3.3.3 配置 PIO class driver（针对 PIP）

如果应用程序使用 PIP 组件 I/O API 函数，应使用连接 mini - driver 的 PIO 适配器。PIP 组件和 PIO 适配器共同创建了一个 class driver，PIP 函数与 PIO 适配器通信，而 PIO 适配器与 mini - driver 通信。此时，在 DSP/BIOS 配置中还不能创建 PIO 对象，创建适当参数的 DIO 对象等价于创建 PIO 对象。

PIP 组件不支持动态创建实例。与其他 DSP/BIOS 组件类似，如果应用程序使用 PIP 模式，PIP 组件的初始化函数会在应用程序启动的时候被 DSP/BIOS 自动调用。

例如，对 PIP 配置步骤如下。

- (1) 为 mini - drivver 创建一个 UDEV。
- (2) 创建一个 SWI 对象。
- (3) 利用 PIP - Buffered Pipe Manager 创建两个 PIP 对象，并对它们重命名。
- (4) 右键单击第一个 PIP，选择 Properties。设置如图 3.37 所示的属性。



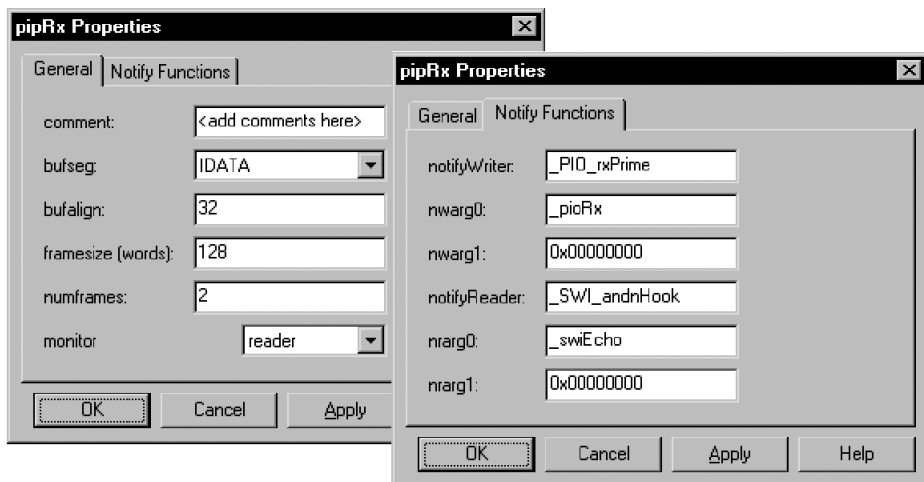


图 3.37 pipRx 属性配置

(5) 右键第二个 PIP，选择 Properties，为 pip Tx 设置如图 3.38 所示的属性。

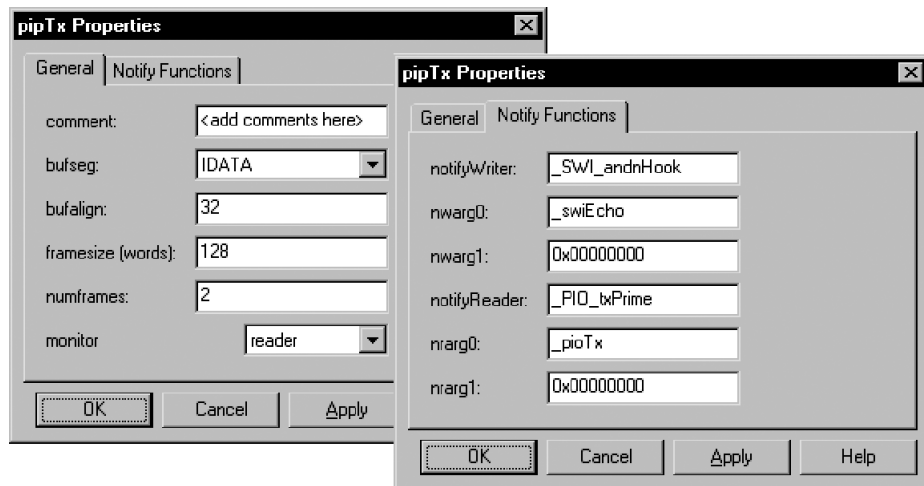


图 3.38 pipTx 属性配置

(6) 使用 PIO 组件 API 函数来动态创建两个 PIO 通道对象 (pioRx 和 pioTx)，例如：

```
void main()
{
    /* Initialize PIO module */
    PIO_init();
    /* Bind PIPs to channels using PIO class drivers */
    PIO_new(&pioRx, &pioRx, " / codec ", IOM_INPUT, NULL);
    PIO_new(&pioTx, &pioTx, " / codec ", IOM_OUTPUT, NULL);
    ...
}
```

(7) 使用适当的 mini-driver 库连接应用程序。

### 3.3.3.4 配置应用程序来使用 GIO Class Driver

GIO 组件配置参数如图 3.39 所示。

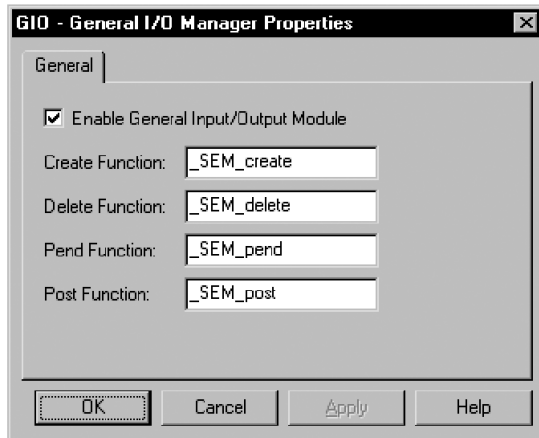


图 3.39 GIO 管理器

GIO 有可以控制 I/O 阻塞时所用的全局参数，默认情况下，GIO 使用 SEM 组件中的旗语对象。GIO 组件不支持建立对象实例，如果应用程序使用 GIO，则在初始化的时候 GIO 函数就会被其自动调用。

例如，对 GIO 配置过程如下。

- (1) 创建 UDEV。
- (2) 使用 GIO 中的 API 函数来动态创建 GIO 通道对象，例如：

```
GIO_Handle input;
Void main() {
    input = GIO_create( " /uart " , IOM_INPUT, &status, NULL, NULL);
    ...
}
```

- (3) 使用应用程序 mini - driver 库来连接你的应用程序。

### 3.3.4 GIO 组件

GIO 组件执行 GIO class driver，这是为应用程序提供一个阻塞（同步）读写的 API。使用多重 IOM mini - driver 的应用程序可以减少整体代码的规模。

GIO 具有如下特性。

- 提供阻塞（同步）读写 API。
- 使用 IOM 接口实现与特定设备 mini - driver 的通信。
- 支持多重设备驱动。
- 支持双向通道。
- 允许模块化函数的用户配置。

- 支持为新领域（比如视频）添加 API。

最后一个特性十分重要，GIO\_submit 函数支持添加自定义的 API 的标准路径。ASYNC 组件就是一个实例。这个组件提供的 DDK 支持使用不阻塞线程的应用程序。这个组件不会增加应用程序的代码大小，它是使用宏调用已有的 GIO 模块来实现的。

设计 GIO class driver 的目的是使代码和数据的规模最小化，同时仍提供同步读写 API 等必要的基本功能。GIO API 可以被连接 IOM mini-driver 的应用程序直接调用。这些 GIO API 是作为 class driver 的。GIO 模块在 DSP/BIOS API 参考手册中有详细描述（C5000 的文献号是 SPRU404，C6000 的文献号是 SPRU403）。

图 3.40 表示了 GIO class driver 与其他系统组件的关系。

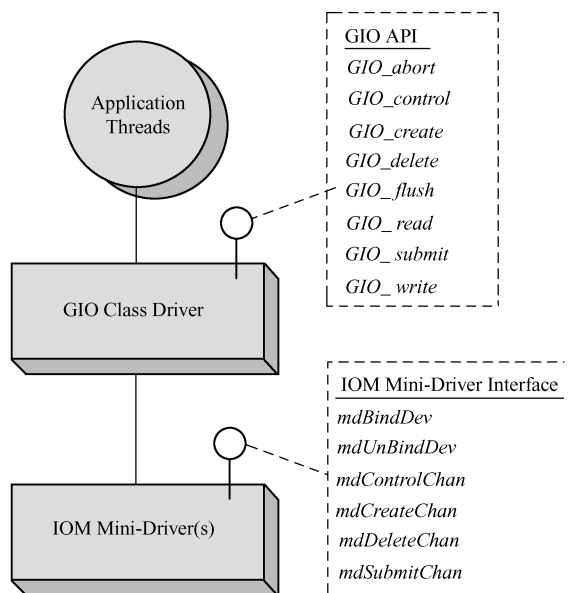


图 3.40 GIO Class Driver 接口

GIO\_create 函数为 IOM 通道实例创建了一个 GIO 对象。一个 GIO\_Obj 有如下结构：

```

typedef struct GIO_Obj {
    IOM_Fxns *fxns;      /* pointer to function table */
    Uns mode;            /* create mode */
    Uns timeout;         /* timeout for blocking */
    IOM_Packet syncPacket; /* for synchronous use */
    QUE_Obj freeList;    /* frames for asynch I/O */
    Ptr syncObj;         /* ptr to synchronization obj */
    Ptr mdChan;          /* ptr to channel obj */
} GIO_Obj, *GIO_Handle;
  
```

GIO 对象为下列重要信息提供存储空间。

- IOM mini-driver 函数表（fxns）。
- 创建通道的模式，包括：IOM\_INPUT，IOM\_OUTPUT，IOM\_INOUT。
- Class driver 和 mini-driver 间的所有 IOM\_Packet（freelist），它们是由 GIO 分配和释

放的。

- 同步对象。
- 指向特殊 IOM mini - driver 通道对象 (meChan) 的指针。

### 3.3.5 DDK 应用举例——Video Port mini - driver

本小节以 DM642 的视频口驱动为例，详述了 DSP/BIOS 配置驱动的方法。

#### 3.3.5.1 概述

Video port mini - driver 是 TI 公司针对视频提供的专门驱动程序套件。它实际上是 IOM mini - driver 的一个组成部分，是双层设备驱动模型中的底层。上层是 FVID 组件，是对 DSP/BIOS GIO class driver 进行了简单封装。GIO 为大量的各种 mini - driver 提供一系列独立 API，FVID 提供了用于视频捕获和显示的 API。

图 3.41 所示是 DM642 视频端口的视频抓取和显示的 mini 驱动的结构。为了提高代码的可重复性，DM642 视频抓取和显示 mini - driver 分成两部分：一个通用部分和一个特殊部分。在其他 mini - driver 中，例如 C6x1 EDMA McBSP mini - driver，class driver 与 mini - driver 中对应不同板卡的部分相连接。在 DM642 视频 mini - driver 中，FVID/GIO 与 mini - driver 中的通用部分相连，与板卡相关的部分通过一个称作外部设备控制器 (EDC) 的接口接入到 mini - driver 的通用部分。

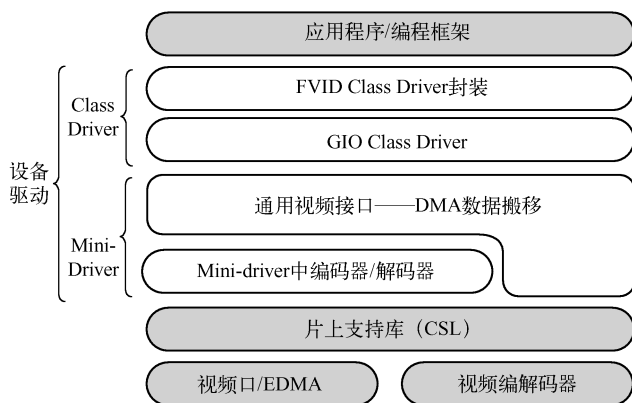


图 3.41 DM642 视频端口的视频抓取和显示的 mini 驱动的结构

驱动的通用部分使用 EDMA 从视频口搬移数据。与板卡相关的代码，在抓取的驱动中初始化、配置了 SAA7115 视频解码；而在显示驱动中初始化、配置了 SAA7105 视频编码。这些 EDC 组件设置了视频编解码器，让视频口抓取或者显示特定的视频数据。

这些与板卡相关的部分同时也需要调用 DM642 EVM 板级支持库 (BSL) 中的 EVM642\_init () 函数来初始化 EVM 和 DSP，包括设置 EMIF、管脚复用和 I<sup>2</sup>C 控制器等。一个应用程序必须连接 3 个库：一个是板卡相关的库，如 SAA7115 或者 SAA7105；第二个是通用 VPORTCAP 或者 VPORTDIS；第三个是 BSL。在抓取端，这三个库命名为 evm642\_saa7115.l64、evm642\_vportcap.l64、evmdm642.l64；在显示端，则为 evm642\_saa7105、evm642\_vportdis.l64、evmdm642.l64。

### 3.3.5.2 Vporrt 配置

打开 DSP/BIOS config 中的 .tcf 文件，在 Input/Output 选项中找到 Device Drivers，用右键选中 User - Defined Devices，插入三个设备，如图 3.42 所示。可以看到有两个 capture 和一个 display，这是因为所使用的板子上面存在两个视频输入接口和一个视频输出口。

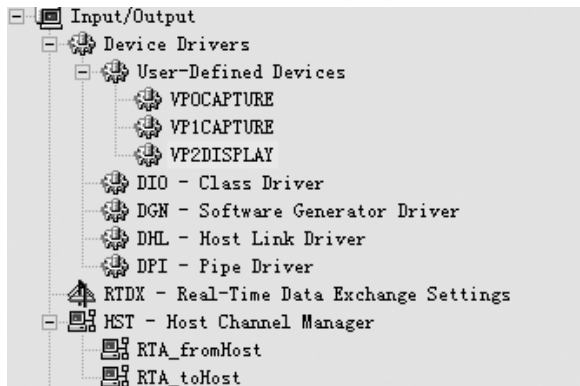


图 3.42 建立 Device Drivers

capture 和 display 配置如图 3.43 所示，两个 capture 具有不同的 device id，VPO CAPTURE 的 id 为 0x00000000，VP1CAPTURE 的 id 为 0x00000002，其他配置都是相同的。



图 3.43 (a) capture 配置情况



图 3.43 (b) display 配置情况

Port 参数设置为一个结构体，如下：

```
typedef struct VPORT_PortParams {
    Int    versionId;
    Bool   dualChanEnab;
    Uns    vc1Polarity;
    Uns    vc2Polarity;
    Uns    vc3Polarity;
    EDC_Fxns * edcTbl[ 2 ];
} VPORT_PortParams;
```

versionId: 驱动版本号。

dualChanEnable: 双通道模式开关 (仅适用 capture)。

vc1Polarity: vctrl1 pin 极性 (类推后 2 个)。

edcTbl[2]: 维数为 2 的 EDC 函数表的指针数组。

整个视频工程是基于任务来完成的, 这里只用到了一个静态任务, 在主函数完成后执行静态任务, 调用函数 tskVideoLoopback()。

下面是程序的基本架构。

```

/* 初始化 主要为句柄的定义、指针的定义以及视频像素的大小 */
FVID_Handle disChan;
FVID_Frame * disFrameBuf;
Int numLinesDis = EVMDM642_vDisParamsChan. imgVSizeFld1;
Int numLinesCap = EVMDM642_vCapParamsChan. fldYStop1 -
    EVMDM642_vCapParamsChan. fldYStrt1 + 1;
FVID_Handle capChan;
FVID_Frame * capFrameBuf;

/* 在堆内存中分配 capture 和 display 缓冲区 buffers */
EVMDM642_vCapParamsChan. segId = EXTERNALHEAP;
EVMDM642_vDisParamsChan. segId = EXTERNALHEAP;
EVMDM642_vDisParamsSAA7121. hI2C = EVMDM642_I2C_hI2C;
EVMDM642_vCapParamsTVP5150A. hI2C = EVMDM642_I2C_hI2C;

/* 初始化 capture 驱动 */
capChan = FVID_create( " /VPOCAPTURE/A/0", IOM_INPUT, &status, (Ptr) &EVMDM642_vCapParamsChan, NULL);

/* 初始化 display 驱动 */
disChan = FVID_create( " /VP2DISPLAY/0", IOM_OUTPUT, &status, (Ptr) &EVMDM642_vDisParamsChan, NULL);

/* 配置视频编码器与解码器 */
FVID_control( capChan, VPORT_CMD_EDC_BASE + EDC_CONFIG, (Ptr) &EVMDM642_vCapParamsTVP5150A);

FVID_control( disChan, VPORT_CMD_EDC_BASE + EDC_CONFIG, (Ptr) &EVMDM642_vDisParamsSAA7121);

FVID_control( capChan, VPORT_CMD_START, NULL);
FVID_control( disChan, VPORT_CMD_START, NULL);
/* 分配视频的空间,即在接收端和发送端开辟 2 个 buffer */
FVID_alloc( capChan, &capFrameBuf);
FVID_alloc( disChan, &disFrameBuf);
/* 释放缓冲区 */
FVID_free( capChan, &capFrameBuf);
FVID_free( disChan, &disFrameBuf);

```



### 3.3.5.3 数据块图表

图 3.44 展示的是显示驱动的顶层数据块图。

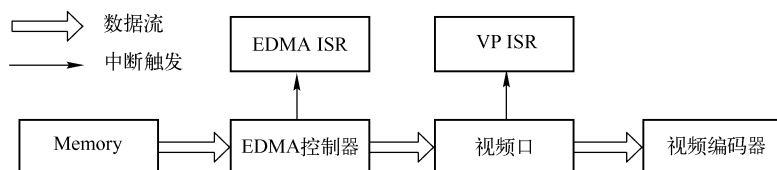


图 3.44 显示缓冲区

在显示过程中，数据通过 EDMA 转换器从内存中的 frame buffer 流入到视频端口 FIFO。反过来，视频端口输出数据到外部视频解码器用于显示。在一个完整的 frame 从内存传入到视频端口后，EDMA 中断被触发。这个中断是必要的，目的如下：

- Frame buffer 管理；
- EDMA 重载入更新；
- 通知 class driver 是否有空的帧缓冲可用，以便应用程序通过调用回调函数填充帧；
- 错误处理或者视频端口同步，可以使能视频端口全局中断。

视频抓取驱动与显示驱动完全类似，只是数据流方向相反。

### 3.3.5.4 buffer 管理

包含视频数据的帧缓冲被驱动分配并且初始化。驱动分配的帧 buffer 的数量是实时配置的，最小为需求的三倍。在分配之前，驱动根据信道配置参数估算每个 buffer 的大小。例如，能够保存整个 NTSC 视频帧的大小为  $720 \times 480 \times 2$ 。

通过使用 FVID\_alloc()、FVID\_free() 和 FVID\_exchange() 函数，帧 buffer 在应用和驱动之间交换。然而，帧管理策略在捕捉和显示驱动中是不同的，如图 3.45 和图 3.46 所示。

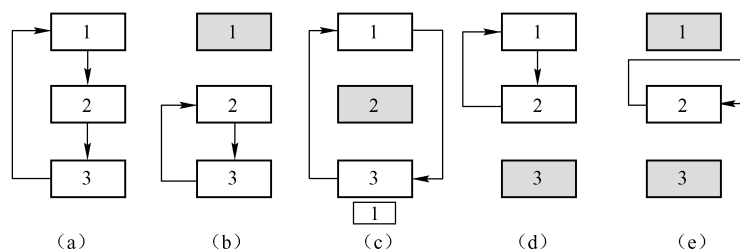


图 3.45 capture 驱动缓冲区管理

在捕捉的时候，所有的 buffer 最初都在空缓冲队列中，构成循环队列，如图 3.45 (a) 所示。

当应用程序调用 FVID\_alloc()，从队列中获取了一个 buffer，并且开始向里面填充数据，其余的 buffer 留在循环队列中，如图 3.45 (a) 到 (b) 所示。

当应用程序调用 FVID\_free()，一个空的 buffer 被应用返回到空缓冲队列。这个过程如图 3.45 从 (b) 到 (a) 或者从 (e) 到 (b)。

当应用程序调用 FVID\_exchange()，一个空的 buffer 被应用返回空缓冲队列，并且一个

带着最新数据的 buffer 传给应用。这等价于相继调用 FVID\_free() 和 FVID\_alloc()。如图 3.45 中从 (b) 到 (c) 和 (c) 到 (d)。

图 3.46 中,  为应用获取的 buffer,  为输出队列 buffer,  为当前 buffer。

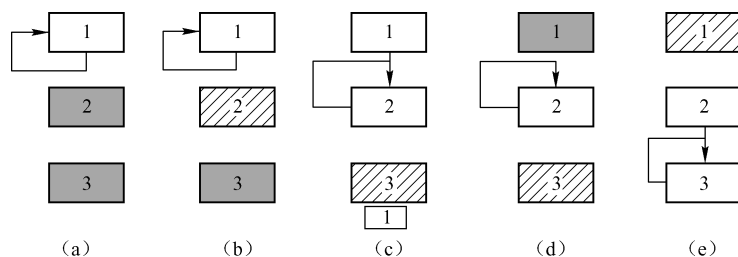


图 3.46 display 驱动缓冲区管理

在显示的时候, 初始化时除了一个 buffer, 其余所有的 buffer 都在输出队列中, 准备被应用程序抓取。驱动重复地显示当前 buffer。在图 3.46 (a) 中显示。

当应用程序调用 FVID\_alloc(), 它从驱动中获取了一个 buffer, 并且开始向里面填充数据, 同时驱动仍然在显示当前 buffer。如图 3.46 (b) 和 (d) 所示。

当应用程序调用 FVID\_free(), 它返回一个准备显示的 buffer 到驱动。相应地, 驱动在它完成显示之前的一个 buffer 后, 设置这个 buffer 作为它的当前 buffer。这个过程如图 3.46 中 (b) 到 (c) 到 (d)。

当应用程序调用 FVID\_exchange(), 它返回一个准备显示的 buffer 给驱动, 并且从驱动要求一个空的 buffer。这个等价于相继调用了 FVID\_free() 和 FVID\_alloc。如图 3.46 中的 (d) 到 (e)。

### 3.3.5.5 Cache 连贯性

确保 Cache 连贯性是应用程序的责任, 因为驱动在这个方面不做任何事情。这是因为为了更快速地访问 CPU, 通常数据被 EDMA 在快速片上 SRAM 和慢速片外 SD-RAM 之间移动。此外, 算法能够运用乒乓 buffer 方案以并行实现 EDMA 传输和 CPU 运行, 因此随着数据移动, 隐藏了大部分的相关开销。如果是这种状况, Cache 更新与清除操作可以避免, 只需将帧 buffer 按 Cache 边界对齐即可。

然而, 如果应用程序直接访问这些 buffer, 则必须自行完成 Cache 更新与清除操作, 来确保 Cache 的连贯性。EDMA 通过 EMIF 直接访问外部存储器, 同时 CPU 通过 Cache 访问数据。

## 3.4 DSP/BIOS LINK

DSP/BIOS LINK 是服务于 GPP 和 DSP 之间通信的基础软件, 它提供了一套通用的 API, 便于应用程序对 GPP 和 DSP 之间物理连接的抽象访问。目的是使用户程序专注于应用本身的发展, 减少对烦琐的处理器间连接的处理。

DSP/BIOS LINK 可以跨平台使用, 既可以用于 GPP 加 DSP 的 SOC, 也可以单独用于 GPP 和 DSP。

正如此软件的名称所示, 在 DSP 上需要运行 DSP/BIOS, 而 GPP 上不需要运行专门的操

作系统。软件发布在一个参考平台上，可以针对一组操作系统。发布的软件包中包括全部的源代码，方便用户移植到指定的平台和操作系统上。

根据其支持的平台和操作系统，DSP/BIOS LINK 提供下列服务：

- 基本的处理器控制；
- 跨多个处理器的共享/同步存储器池；
- 用户事件通知；
- 共享数据结构的互斥访问；
- 基于数据流的链表；
- 逻辑通道上的数据传输；
- 基于 DSP/BIOS 的 MSGQ 模块的消息；
- 基于数据流的环形缓存器；
- 零拷贝消息。

一个典型的应用程序不一定会用到 DSP/BIOS LINK 的全部服务，可能只用了 GPP 和 DSP 间传递消息这一机制。在这种情况下，DSP/BIOS LINK 可以在编译时进行裁剪，只包括需要的模块。

### 3.4.1 DSP/BIOS LINK 的软件结构

DSP/BIOS LINK 的软件结构如图 3.47 所示。

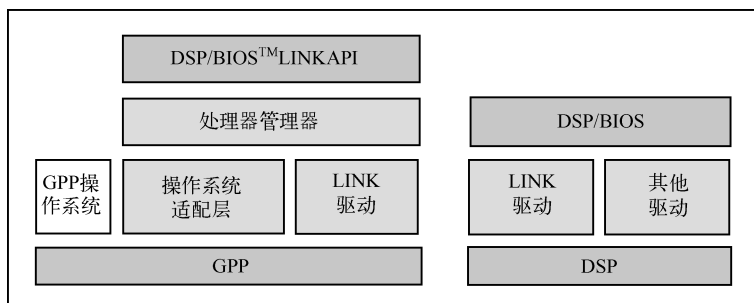


图 3.47 DSP/BIOS LINK 的软件结构

### 3.4.1.1 GPP 端

在 GPP 端，假定已经运行了一个操作系统（GPP OS）。还需要以下四个组件。

(1) 操作系统适配层 (OS ADAPTATION LAYER), 封装了 DSP/BIOS LINK 的其他组件需要调用的操作系统服务。该适配层提供一个通用的 API, 将其他组件和操作系统隔离。所有的其他组件都使用这个 API 访问操作系统, 而不是直接使用系统调用, 从而使得 DSP/BIOS LINK 可以迁移到不同的操作系统上。

(2) LINK 驱动 (LINK DRIVER), 封装了 GPP 和 DSP 之间的物理连接的底层控制操作。该模块控制 DSP 的运行, 以及用定义好的协议进行跨越 GPP 和 DSP 边界的数据传输。

(3) 处理器管理器 (PROCESSOR MANAGER) 维护所有组件的信息, 允许系统中添加不同的引导装载程序。它通过 API 层向用户提供 LINK DRIVER 的控制操作。

(4) DSP/BIOS LINK API, 是 GPP 上所有客户端的接口。这是一个很小的组件, 除了确认参数外, 并不做更多的处理。API 层可以看成是“皮肤”, “肌肉”则包括在 PROCESSOR MANAGER 和 LINK DRIVER 里。这个薄 API 层使 DSP/BIOS LINK 很容易地与 Linux 之类的特定操作系统内核隔离; 而在其他一些操作系统上这样的隔离是没有必要的。

### 3.4.1.2 DSP 端

LINK DRIVER 是 DSP/BIOS 的驱动之一, 专门用于与 GPP 在物理链路上的通信。

在 DSP 上, 没有专门的 DSP/BIOS LINK API。通信 (数据/消息的传输) 是用 DSP/BIOS 的 SIO/ GIO/ MSGQ 等模块来完成的。

## 3.4.2 DSP/BIOS LINK 的关键组件

### 3.4.2.1 PROC

PROC 组件在应用程序空间中代表 DSP 处理器, PROC 是“处理器”的缩写。

该组件所提供的服务有:

- 初始化 DSP, 使 GPP 可以访问 DSP;
- 向 DSP 装入代码;
- 从可执行文件所指定的地址开始执行;
- 读写 DSP 存储器;
- 停止执行;
- 和特定 DSP 平台相关的其他控制操作;
- 对给定的符号, 取得 DSP 的地址。

API 可以支持多个 DSP, 所以需要传入 processorId 参数标明是哪个 DSP。

### 3.4.2.2 POOL

POOL 组件提供了 API 用于配置跨处理器的共享内存区域, 还提供了两个 CPU 间的缓存数据同步的 API 接口。

DSP/BIOS LINK 的其他模块会使用这些缓冲区实现处理器间的通信功能。

该模块所提供的服务有:

- 通过打开和关闭调用来配置共享的存储器区域;
- 在共享的存储器区域内分配和释放缓冲区;
- 在不同的地址空间内对缓冲区地址进行转换 (例如, 从 GPP 到 DSP);
- 在不同 CPU 核之间实现内存数据的同步。

此组件负责对不同存储器池提供一个统一的规范, 可以根据 DSP/BIOS LINK 运行的具体硬件平台或操作系统来实现。此组件是基于 DSP/BIOS 的 POOL 接口实现的。

### 3.4.2.3 NOTIFY

NOTIFY 组件用于应用程序注册远程处理器上所发生的事件, 并将事件通知发给远程处理器。应用程序登记一个回调函数, 此回调函数带有一个参数, 与远程处理器的事件相关。

于是,应用程序可以向远程处理器发送特定的事件通知,以及与此事件相关的数值。

NOTIFY 组件为事件通知定义了优先级,优先级通过事件号定义,低编号的事件代表更高的优先级。

如果某事件通知不再需要使用,应用程序也可以实时注销相应的回调函数。

#### 3.4.2.4 MPCS

应用程序通过 GPP 和 DSP 之间的 MPCS (多处理器临界区),互斥访问共享数据结构。

应用程序有时候需要定义属于自己的,并能够被多个处理器访问的数据结构,用于多个处理器之间通信的消息分片。但是,应用程序必须保证多个处理器,或者每个处理器上的多个任务之间互斥地访问这些数据结构,从而保证数据的一致性。MPCS 组件就是用来实现此功能的。在拥有共享内存区的多处理器系统中,可以实现 GPP 和 DSP 之间的 MPCS。

MPCS 组件提供了 API 接口来创建和删除 MPCS 实例,每个 MPCS 实体都通过一个系统唯一的字符串名字来标识。每一个需要使用 MPCS 的客户端都需要调用 API 打开函数来获取句柄。当不再需要使用 MPCS 时,通过相应的 API 函数来关闭句柄。MPCS 也提供了进入和离开临界区的 API 函数。

如果由用户提供 MPCS 对象要求的存储空间,则必须从所有处理器都可以访问到池中分配的此存储空间;如果在创建对象的时候没有提供存储空间,则 MPCS 对象用指定池的 ID 号在内部分配空间。

#### 3.4.2.5 MPLIST

MPLIST 组件提供 GPP 和 DSP 之间传输机制的双循环链接的链表。在 GPP 和 DSP 之间存在共享存储空间的设备上,该模块实现共享存储空间的链表。对于不存在共享存储空间的设备,该模块内部保持和远程处理器上链表的一致性。

该组件提供了创建和删除 MPLIST 实例的 API 函数,每个 MPLIST 实例都通过一个系统唯一的字符串名字来标识。每一个需要使用 MPLIST 的客户端都需要调用 API 打开函数来获取句柄。当不再需要使用 MPLIST 时,通过相应的 API 函数来关闭句柄。

MPLIST 组件提供了一系列 API 函数可供使用。这些 API 函数包括:删除列表中第一个元素,在列表最后加入一个元素,或者在任意位置加入或删除元素,判断表是否为空。通过指向首元素的指针遍历列表,以得到任意指定元素的下一个元素。

#### 3.4.2.6 CHNL

CHNL 组件描述应用空间的一个逻辑数据传输通道,负责 GPP 与 DSP 之间的数据传输。通道 channel 的概念如下:

- 一种 GPP 与 DSP 间传输数据的方式;
- 一个 GPP 与 DSP 物理连接上的逻辑实体映射;
- 通过唯一的数字标识连接到 DSP 的一组通道中的某个物理连接;
- 通道是单向的,通道方向由 API 决定。

DSP 与 GPP 的一条物理连接可以被多个通道复用,是否复用取决于相关链接驱动是否支持。通道传输的数据中不包含目的地和源的任何信息。每一侧的处理器都可以是生产者或消费者,必需明确地建立数据通路。CHNL 组件采用发送 - 回收 (issue - reclaim) 模型进行

数据传输，它模仿了 DSP/BIOS 的 SIO 模块的发送 - 回收模型。

### 3.4.2.7 MSGQ

MSGQ 组件实现基于消息的队列，负责 GPP 与 DSP 的可变长度的短消息交互，基于 DSP/BIOS 的 MSGQ 模块实现。

消息的发送接收都通过消息队列实现，读者从消息队列接收消息，而写者将消息写入到消息队列中；一个消息队列只可以有一个写者，但可以有多个读者。一个任务可以读写多个消息队列。

如果用户期望收到消息，就必须创建消息队列。在消息发出之前，必须明确消息的接收者。

### 3.4.2.8 RINGIO

RINGIO 组件提供基于数据流的循环缓冲区。该组件允许在共享存储空间创建循环缓冲区，此缓冲区的读者和写者可以是不同的处理器。

每个 RINGIO 实例拥有一个读者和一个写者。写者从数据缓冲区获取空缓冲区，当写者填入数据后，将缓冲区提交到共享存储。读者从数据缓冲区获取有效数据，当数据读取后，相应存储空间的数据就标记为无效，成为空缓冲区。

RINGIO 组件也提供数据标志等属性同步传输的 API 函数。如：EOS（流结束标记）、时间戳、流偏移地址等。

## 3.4.3 典型的应用流程

本节介绍了各个组件的初始化、执行和结束阶段的典型步骤。由于组件间相互的依赖关系，一个组件的初始化可能会取决于另外一个组件的执行。因此，编写应用程序的时候，必须要考虑组件之间的这些依赖关系。

### 3.4.3.1 初始化

本小节简述了每个组件初始化阶段的各个步骤。这些步骤完成必要的资源分配和适当的初始化。

#### 1. PROC

典型的过程如下。

- 完成组件的基本初始化，包括初始化延伸到较低层组件的序列和填充所需的数据结构。
- 与特定的 DSP 关联通信。在这个过程中，较低层的组件初始化 DSP 的硬件接口，以便 GPP 能访问 DSP。
- 对 DSP 装载可执行文件。这个可执行文件包括在 DSP 上的应用程序。

第一个连接到 DSP 的客户端将控制 DSP。这种控制模式是必须的，这样其他客户不会对 DSP 造成影响，比如停止 DSP 或者加载其他可执行文件。

使用的 API 包括：PROC\_setup(), PROC\_attach() 和 PROC\_load()。

#### 2. POOL

典型的过程如下。



在分配的数据缓冲区或消息中打开池。默认的池和 DSP/BIOS LINK 关联如下。

- SMAPOOL: 在共享内存中为跨处理器访问分配零拷贝缓冲区。
- BUFPOOL: 用于固定大小的缓冲区。

该缓冲区可以利用池中的 ISR 和 DPC 来分配和释放。

每个池必须初始化一次。多个使用相同池功能表,但具有不同 ID 的池,可以使用静态配置工具进行配置,而且每个池都必须初始化一次。

使用的 API 为 POOL\_open()。

### 3. NOTIFY

典型的过程如下。

为远程处理器上的事件通知登记一个回调函数。登记的固定参数可以随意指定,当收到事件通知时,参数也与回调函数一起收到。

使用的 API 为 NOTIFY\_register()。

### 4. MPCS

典型的过程如下。

- 创建一个全系统唯一命名的 MPCS 实例。MPCS 对象的属性中包括一个共享存储空间,此空间可由用户提供,否则可基于 POOL 的 ID 自动分配。
- 通过名字打开 MPCS 得到控制部分,这样可以进一步调用 MPCS 组件。

使用的 API 为 MPCS\_create() 和 MPCS\_open()。

### 5. MPLIST

典型的过程如下。

- 创建一个全系统唯一命名的 MPLIST 实例。MPLIST 对象的属性中包括一个共享存储空间,此空间可由用户提供,否则可基于 POOL 的 ID 自动分配。
- 通过名字打开 MPLIST 得到控制部分,这样可以进一步调用 MPLIST 组件。

使用的 API 为 MPLIST\_create() 和 MPLIST\_open()。

### 6. CHNL

典型的过程如下。

- 在 GPP 和 DSP 之间创建数据传输通道。
- 为信道中传输的数据分配缓冲区。
- 在启动数据传输之前准备好数据缓冲区。

应用程序必须决定用哪些信道传输数据。在 GPP 和 DSP 之间信道必须以正确的方向打开,从而保证数据正常传输。

使用的 API 为 CHNL\_create() 和 CHNL\_allocateBuffer()。

### 7. MSGQ

典型的过程如下。

- 打开一个消息队列,所有用于客户端的消息将被添加到该队列。
- 打开一个消息队列,异步错误信息将被排队。这个队列可以与上一步中创建的相同。
- 打开通向 DSP 的消息传输。
- 通过名字查找传输消息的远程队列。

需要注意的是,最后一步可能需要获得来自 DSP 的响应。因此,必须在 DSP 运行后才能被执行。消息队列是通过全系统唯一名称来标识的。

使用的 API 包括：MSGQ\_transportOpen()、MSGQ\_open()、MSGQ\_setErrorHandler() 和 MSGQ\_locate()。

## 8. RING IO

典型的过程如下。

- 创建一个全系统唯一命名的 RINGIO。其数据缓冲区、属性缓冲区、控制结构和锁结构的 ID 属性都是基于 POOL ID 提供的。
- 打开 RINGIO 读者或写者模式。返回一个客户具体应用的控制，这将用于进一步调用 RINGIO 组件。
- 如果需要，可对 RINGIO 客户端设置通知函数。有指定的通知类型，以及是否 RINGIO 已打开读者或写者模式，当缓冲区空或满的标记到达后，会自动调用通知函数。

使用的 API 包括：RingIO\_create()、RingIO\_open() 和 RingIO\_setNotifier()。

### 3.4.3.2 执行

本小节简述了每个组件执行阶段的各个步骤。

#### 1. PROC

典型的过程如下。

- 开始执行早已加载在 DSP 上的可执行文件。DSP 开始执行后，就不再依赖 PROC 组件。
- 读 DSP 存储器。
- 写 DSP 存储器。
- 应用程序完成后，停止运行。

使用的 API 包括 PROC\_start()、PROC\_read()、PROC\_write() 和 PROC\_stop()。

#### 2. POOL

典型的过程如下。

- 从池中分配一个缓冲区。
- 如果需要，将在其他地址空间（用户、内核、物理和 DSP）分配缓冲区。
- 释放以前池中分配的缓冲区。

相关的 API 包括 POOL\_alloc()、POOL\_translateAddr() 和 POOL\_free()。

#### 3. NOTIFY

典型的过程如下。

- 与一个可选设备的值一起，发送一个事件通知到远程处理器。
- 接收一个远程处理器的事件通知。通过登记时指定的固定参数，调用回调函数。通过事件接收可选设备值。

相关的 API 是 NOTIFY\_notify()。

#### 4. MPCS

典型的过程如下。

- 输入指定的关键语句，获得 MPCS 保护共享结构入口。
- 在 MPCS 保护的共享数据结构上的操作完成后，离开 MPCS，让其他处理或处理器可以使用。

使用的 API 有 MPCS\_enter() 和 MPCS\_leave()。

## 5. MPLIST

典型的过程如下。

- 在链表尾部，从池中分配一个缓冲区。
- 从链表的头部删除缓冲区。
- 如果需要的话，检查列表是否为空。
- 如果需要的话，在表中存在的元素前插入一个缓冲区。
- 如果需要的话，通过断开链接可以移除链表中的指定元素。
- 如果需要的话，可以得到指向链表第一个元素的指针。
- 如果需要的话，可以得到指向链表特定元素的指针。

使用的 API 有 `MPLIST_putTail()`、`MPLIST_getHead()`、`MPLIST_isEmpty()`、`MPLIST_insertBefore()`、`MPLIST_removeElement()`、`MPLIST_first()` 和 `MPLIST_next()`。

## 6. CHNL

典型的过程如下。

- 对已创建的通道分配缓冲区。为输出通道分配一个填充好的缓冲区，DSP 上的远程客户端接收此缓冲区。为输入通道分配一个空的缓冲区，用于接收 DSP 上远程客户端发送的数据。
- 在通道上，对上一步分配的缓冲区回收。这是一个同步操作，即客户端一直被阻塞，直到输入输出操作成功完成（或者超时）。

使用的 API 有 `CHNL_issue()` 和 `CHNL_reclaim()`。

## 7. MSGQ

典型的过程如下。

- 使用池分配一个消息。
- 在消息队列中发送消息。
- 从消息队列中接收消息。
- 从接收消息中得到源消息队列的句柄。这个消息句柄用来回复接收到的消息。
- 如果需要，可以从消息中获取信息，也可以在消息中设置一些信息，或者得到一个消息队列的信息。
- 释放消息。

使用的 API 有 `MSGQ_alloc()`、`MSGQ_put()`、`MSGQ_get()`、`MSGQ_getSrcQueue()`、`MSGQ_getMsgId()`、`MSGQ_getMsgSize()`、`MSGQ_setMsgId()`、`MSGQ_getDstQueue()`、`MSGQ_setSrcQueue()`、`MSGQ_isLocalQueue()` 和 `MSGQ_free()`。

## 8. RING IO

典型的过程如下。

- 从 RINGIO 中获取缓冲区。
- 如果是以写者模式打开 RINGIO，在获得的缓冲区中，根据应用需求可以设置一个固定或者可变属性的偏移量。如果是以读者模式打开 RINGIO，当一个属性是目前读偏移，可以得到这个固定或者可变的属性。
- 如果是以写者模式打开 RINGIO，写入获取的空缓冲区，释放已经初始化的缓冲区。如果是以读者模式打开 RINGIO，读取获得的全部缓冲区，释放已经读取的缓冲区。

- 如果需要，可以得到 RINGIO 客户端的当前状态信息。

使用的 API 有 RingIO\_acquire()、RingIO\_setAttribute()、RingIO\_getAttribute()、RingIO\_setvAttribute()、RingIO\_getvAttribute()、RingIO\_cancel()、RingIO\_release()、RingIO\_flush()、RingIO\_getValidSize()、RingIO\_getEmptySize()、RingIO\_getAcquiredOffset()、RingIO\_getAcquiredSize() 和 RingIO\_getWatermark()。

### 3.4.3.3 结束

本小节简述了每个组件结束阶段的各个步骤。这些步骤确保所有以前分配的资源能正确地释放。

#### 1. PROC

典型的过程如下。

- 从 DSP 中分离。如果客户是 DSP 的拥有者（即第一个在 DSP 上运行的任务），那么这一步操作会使 DSP 结束。
- 释放初始化阶段分配的资源。

使用的 API 有 PROC\_detach() 和 PROC\_destroy()。

**注意：**PROC 是应用程序最后一个完成的组件。

#### 2. POOL

典型的过程就是关闭池。使用的 API 为 POOL\_close()。

#### 3. NOTIFY

典型的过程如下。

利用登记过程中指定的固定参数，注销回调函数。在这之后，没有接收到该事件的进一步通知。使用的 API 为 NOTIFY\_unregister()。

#### 4. MPCCS

典型的过程如下。

- 关闭早期得到的 MPCCS 句柄。在这之后，没有进一步的调用运行 MPCCS 的 API。
- 删除早期建立的 MPCCS 实例。

使用的 API 包括 MPCCS\_close() 和 MPCCS\_delete()。

#### 5. MPLIST

典型的过程如下。

- 关闭早期得到的 MPCCS 句柄。在这之后，没有进一步的调用运行 MPLIST 的 API。
- 删除早期建立的 MPLIST 实例。

使用的 API 包括 MPLIST\_close() 和 MPLIST\_delete()。

#### 6. CHNL

典型的过程如下。

- 释放初始化步骤时分配的缓冲区。
- 删除通道。

使用的 API 包括 CHNL\_freeBuffer() 和 CHNL\_delete()。

#### 7. MSGQ

典型的过程如下。

- 释放在远程消息队列。

- 关闭远程传输。
- 关闭本地消息队列。

使用的 API 包括 MSGQ\_release()、MSGQ\_transportClose() 和 MSGQ\_close()。

## 8. RING IO

典型的过程如下。

- 关闭 RING IO。
- 删除 RING IO。

使用的 API 包括 RingIO\_close() 和 RingIO\_delete()。

## 3.4.4 使用 DSP/BIOS LINK

### 3.4.4.1 配置开发环境及构建代码

在编译链接 DSPLINK 代码前，需要为用户的编译链接环境和应用配置 DSPLINK 编译链接系统。

这包括下面三个主要步骤。

(1) 按用户要求定制 DSPLINK 的 make 系统。按照安装手册建立编译链接系统。为了定制用户编译链接环境的 make 系统，可能需要对一些文件进行修改。

(2) 建立编译链接环境。提供脚本来建立 DSPLINK 必需的环境变量。

(3) 配置 DSPLINK 的编译链接。执行编译链接配置脚本，为各种参数（如：平台、GPP OS、编译链接配置等）配置 DSPLINK。配置 DSP/BIOS™ LINK 编译链接是一个交互式的过程。在编译链接过程中，包括适当的配置文件。配置编译链接又取决于环境变量（前面提到的通过脚本执行来设置 DSPLINK）

使用由 DSPLINK 发布版提供的通用 make 系统，对 GPP 和 DSP 端的 DSPLINK 代码、例程和测试套件编译链接。make 系统支持在 Linux 和 Windows 系统上的编译链接。

详见《DSP/BIOS Link UserGuide》

### 3.4.4.2 传递参数与动态配置 DSP/BIOS Link

#### 1. 参数传递

GPP 通过 API 函数 PROC\_Load()，把参数传递给 DSP 的 main() 函数。在参数写入 DSP 存储空间前，此 API 函数填充 “.args” 段所在的缓冲区。BIOS 使用 “.args” 段将参数传递到 DSP 的 main() 函数。

在 DSP 可执行文件编译期间，建立 “.args” 段。根据传输到 DSP 的参数，编译器需要建立足够大的段，以避免覆盖段。

下面描述实现方法。

- GPP 端传输参数

下面代码描述了使用 API 函数 PROC\_Load() 实现传输参数的方法。

```
UInt32   argc = 0;
Char8    * argv [ NUM_ARGS ];
```

```

argc = NUM_ARGS;

argv [0] = arg_string_1;
argv [1] = arg_string_2;
...
...
argv [NUM_ARGS - 1] = arg_string_end;

status = PROC_Load (dspID, dspExecutableFileName, argc, argv);

```

### • DSP 端接收参数

下面这行程序加到 tcf 文件，用来建立指定大小（单位为 byte）的 “.args” 段。

```

prog.module(“MEM”) . ARGSSIZE = <number of bytes>

```

## 2. 动态配置

DSPLINK 的静态配置是通过文本配置文件（CFG\_<PLATFORM>.TXT）在早期完成的。

文本配置文件是在 DSPLINK 编译链接期间处理的，它为 GPP 和 DSP 端的 DSPLINK 产生配置需要的头文件和源文件。这些产生的文件随着 DSPLINK GPP 端的内核模块和 DSP 端的 dsplink 库一起编译。

对于发布版 1.40.03，已经用预定义的 C 文件替代文本配置文件。这个 C 文件使用固定的格式来预定义配置参数。这会使用 DSPLINK 默认的用户库来编译。

动态配置 DSPLINK 是通过配置项完成的，这些配置项只允许 GPP 端上的用户提供给 DSPLINK。GPP 端的内核模块和 DSP 端的库不需要重新编译链接。

配置中的改变：通过一个可选的指针修改 PROC\_setup() 的配置结构，此结构与提供的配置原文件是相同的格式。如果提供了有效的指针，则使用应用程序提供的配置值；若没有提供有效指针，则使用缺省/默认值。这确保了现有应用程序的向后兼容性。因为不会产生配置源或头文件，所以 GPP 端的内核模块和 DSP 端的 DSPLINK 库不需要重新编译链接。

使用动态配置的方法。

- 文件 \$(DSPLINK)\config\all\CFG\_<Platform>.c 通过了 dsplink 默认的配置。为了改变配置，用户可以在自己的应用程序代码中拷贝这个文件。
- 根据用户自己的需求改编 <application\_path>\CFG\_<platform> 文件。
- 为了避免和默认配置的冲突，把 LINKCFG\_config 重新命名为 LINKCFG\_appConfig。
- 在用户的应用程序中，添加 <application\_path>\CFG\_<platform> 文件到文件列表进行编译。
- 对用户自己的具有外部声明的 LINKCFG\_appConfig 结构改编，并将其作为一个参数传递给 PROC\_setup。

这使得任何动态配置的改变只需要重新编译链接应用程序，不需要编译链接 dsplink 代码库。



### 3.4.5 应用举例

本节通过八个例子分别描述 DSPLINK 各组件的协同工作，帮助 DSP 应用程序开发者理解 DSPLINK 的工作机制。

#### 3.4.5.1 LOOP

LOOP 示例描述了 DSPLINK 基本数据流的概念，例程演示 GPP 端的任务与 DSP 端的任务之间进行数据传输，DSP 端应用程序使用带 SIO 的 TSK 和带 GIO 的 SWI。LOOP 例程中数据流向如图 3.48 所示。

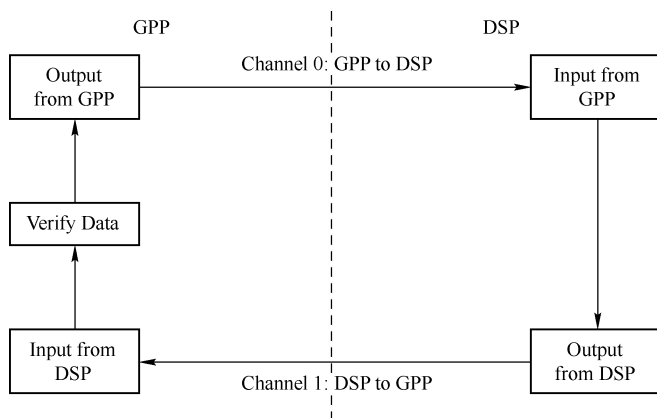


图 3.48 LOOP 例程数据流向

#### 1. 在 GPP 端

##### • 初始化

(1) 客户端为访问 DSP 建立必要的数据结构。这样客户端可以通过 ID - PROCESSOR 接入 DSP。

(2) 打开池分配数据缓冲转移缓冲区。

(3) 在 DSP 上装载 DSP 可执行文件 (loop.out)。

(4) 创建 CHNL\_ID\_INPUT 和 CHNL\_ID\_OUTPUT 数据传输通道。

(5) 在这些通道上为数据传输分配指定大小的缓冲区并填写初值。

##### • 执行

(1) 客户端开始在 DSP 上运行。

(2) 输出缓冲区填满采样数据。

(3) 在 CHNL\_ID\_OUTPUT 发布缓冲区并一直等待回收。

(4) 当完成回收运行，表明缓冲区已通过物理链路传输。

(5) 在 CHNL\_ID\_INPUT 发布缓冲区并一直等待回收。

(6) 一旦缓冲区被回收，其内容会和 CHNL\_ID\_OUTPUT 发布的比较。因为这是一个循环应用，内容应该是一样的。

(7) 客户端重复第 3 步到第 6 步，重复次数由用户指定。

(8) 停止 DSP 操作。

- 结束

- (1) 客户端释放分配的数据传输缓冲区。
- (2) 删除 CHNL\_ID\_INPUT 和 CHNL\_ID\_OUTPUT 通道。
- (3) 关闭池。
- (4) 在 ID\_PROCESSOR 中脱离并释放 PROC 组件。

## 2. 在 DSP 端

使用带 SIO 的 TSK。

- 初始化

- (1) 在 main() 函数中创建客户任务 tskloop。
- (2) 在 main() 函数中配置缓冲区的大小和数量, 用池来分配数据传输缓冲区。
- (3) 建立数据传输的 SIO 通道。
- (4) 为数据传输分配缓冲区并填写初值。

- 执行

- (1) 该任务给输入通道分配一个空的缓冲区并一直等待回收。
- (2) 给输出通道分配一个空的缓冲区并一直等待回收。
- (3) 当完成回收运行, 表明缓冲区已通过物理链路传输。
- (4) 重复这些步骤直到 DSP 所有执行文件完成。

- 结束

- (1) 释放用于数据传输的缓冲区。
- (2) 删除 INPUT\_CHANNEL 和 OUTPUT\_CHANNEL 通道。

使用带 GIO 的 SWI。

- 初始化

- (1) 在 main() 函数中, 建立用于数据输入输出的 GIO 通道。
- (2) 在 main() 函数中配置缓冲区的大小和数量, 用池来分配数据传输缓冲区。
- (3) 创建一个 SWI 对象用于数据传输。SWI 对象的其中一个属性是回调函数 (loop-backSWI)。当 SWI 用于完成在 GIO 通道上的读写请求时, 这个函数被调用。
- (4) 为数据传输分配缓冲区并填写初值。

- 执行

- (1) 在输入数据缓冲区上启动数据传输读请求, 该请求被提交到 INPUT\_CHANNEL 通道上。
- (2) 一旦发布 SWI, 输入缓冲区的内容复制到输出缓冲区。
- (3) 空的缓冲区被重新发布到输入通道, 满的缓冲区被发布到输出通道。
- (4) 在完成两次请求后, SWI 被再次发布。
- (5) 重复第 2 到第 4 步, 直到定时 GPP 应用程序发送到缓冲区。

- 结束

在这个例程中, 因为读写请求, SWI 会不停地发布。所以它永远不会终止。终止序列应该是:

- (1) 释放用于数据传输的缓冲区;
- (2) 删除 GIO 输入输出通道。

## 3. 启动应用程序

这个 loop 例子带有下面的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	./loop. out
缓冲器大小	1 024
迭代次数	10 000

### 3.4.5.2 MESSAGE

MESSAGE 例程描述了 DSPLINK 消息传输的概念，演示了 GPP 端任务与 DSP 端任务之间消息的传递。在 DSP 端，这个应用程序描述了带 MSGQ 的 TSK 和 SWI。MESSAGE 例程的消息流如图 3.49 所示。

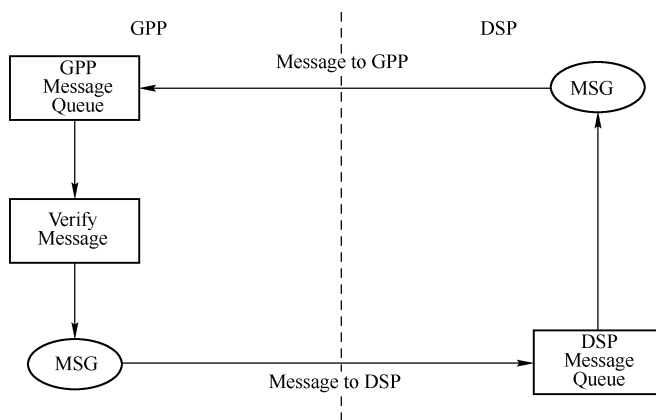


图 3.49 MESSAGE 例程的消息流

#### 1. 在 GPP 端

##### • 初始化

- (1) 客户端为访问 DSP 建立必要的数据结构。
- (2) 通过 ID\_PROCESSOR 接入 DSP。
- (3) 根据用于数据驱动的物理链路，打开池配置信息。
- (4) 打开本地处理器命名的消息队列。
- (5) 设置开放队列用作错误信息处理。
- (6) 在 DSP 上装载 DSP 可执行文件 (message. out)。
- (7) 客户端在 DSP 上启动执行。
- (8) 打开远程传输。
- (9) 一直查找 DSP 端打开的队列。如果查找不成功 (DSP 队列还没有打开)，休眠一段时间后会继续查找队列。

##### • 执行

- (1) 客户端在本地队列获取消息，这个操作会一直进行。
- (2) 一旦接收到消息，会验证消息内容的有效性。
- (3) 将相同的消息传回 DSP 消息队列。
- (4) 收到消息后，其内容与序列号比较，如果是递增的则说明成功。
- (5) 客户端重复第 2 步到第 4 步，重复的次数由用户指定。可以重复无穷多次。

(6) 对有限次循环, 当收到最后一次操作后, 释放消息。

- 结束

- (1) 客户端在 DSP 上释放远程消息队列。
- (2) 关闭远程传输。
- (3) 停止 DSP 程序。
- (4) 重新设置初始阶段设置的错误处理程序。
- (5) 关闭当地消息序列。
- (6) 关闭池。
- (7) 在 ID\_PROCESSOR 中脱离。
- (8) 释放 PROC 组件。

## 2. 在 DSP 端

使用带 MSGQ 的 TSK。

- 初始化

- (1) 通过全局变量 POOL\_config, 池可以静态配置消息。
- (2) 通过全局变量 MSGQ\_config, 可以静态配置 MSGQ 组件。
- (3) 在 main() 函数中创建客户端任务 tskMessage。
- (4) 通过特定的名字, 在本地处理器中打开消息队列。
- (5) 设置开放队列用作错误信息处理。
- (6) 一直查找 GPP 端打开的队列。如果查找不成功 (GPP 序列还没有打开), 休眠一段时间后会继续查找队列。查找操作是同步的。

- 执行

- (1) 任务从池中分配一个消息。
- (2) 发送一个消息到之前查找到的 GPP 消息队列。
- (3) 任务设法从本地队列得到消息。这个获取操作会一直等待。
- (4) 这些步骤重复的次数由用户指定。可以重复无穷多次。

- 结束

- (1) 客户在 DSP 端发送远程消息。
- (2) 重新设置初始阶段设置的错误处理程序。
- (3) 关闭本地消息队列。

使用带 MSGQ 的 SWI。

- 初始化

- (1) 在 main() 函数中调用函数 SWIMESSAGE\_create()。
- (2) 为消息传递建立 SWI 对象。回调函数 messageSWI 是 SWI 对象的一个属性。
- (3) 通过特定的名字, 在本地处理器中打开消息队列。SWI 对象作为消息队列中接收消息的通知对象。这样就保证消息队列中每次接收到一个消息, SWI 都被投递。
- (4) 设置开放队列用作错误信息处理。
- (5) 最后发布 SWI, 用于应用程序的执行阶段。
- (6) 查找 GPP 端产生的序列。这个查找操作是异步的。

- 执行

- (1) 当 SWI 消息被第一时间投递, 客户端查找 GPP 端打开的消息序列。查找操作是异

步的。

(2) 无论何时在 DSP 消息队列接收到一个消息，都会投递 SWI 消息。接收到第一个消息表明异步查找完成。异步查找消息的 ID 是 MSGQ\_ASYNCLOCATEMSGID。收到这个消息，SWI 为正在传输的 GPP 消息序列设置句柄并释放接收到的消息。

(3) 然后分配一个新消息，将其送到 GPP 初始化消息队列。

(4) 随后每个投递的 SWI 表明一个新消息被收到。SWI 送回相同消息到 GPP 队列。

(5) 如果接收到的消息的 ID 是 msgq\_asyncerrormsgid，表明发生了一个错误。通过收到的错误消息的内容，可以确定错误类型。

• 结束

在 message 例程中，每当一个消息在本地消息队列中准备好，SWI 就不断发布。所以它永远不会结束。结束序列步骤：

- (1) 客户在 DSP 上释放远程消息队列；
- (2) 重新设置创建阶段的错误处理程序；
- (3) 关闭本地消息队列。

3. 启动应用程序

这个 message 例子带有下面的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	./message.out
迭代次数	10 000

3.4.5.3 SCALE

这个例程描述了 DSP/BIOS LINK 中数据流组合及消息的概念。它不仅实现了 GPP 任务和 DSP 任务之间的数据传输，也实现了从 GPP 端发送消息到 DSP 端。

在 DSP 端，这个应用描述了使用 SIO 和 MSGQ 的 TSK，以及使用 GIO 和 MSGQ 的 SWI。

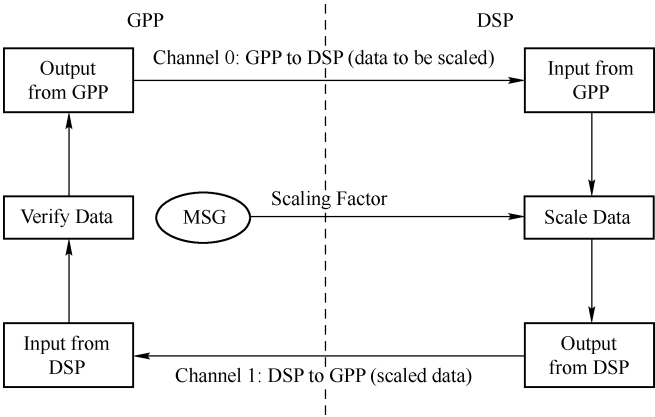


图 3.50 例程应用中的数据 and 消息流

## 1. 在 GPP 端

### • 初始化

- (1) 客户端调用 API, 使 DSP 可接入。
- (2) 根据用于数据驱动的物理链路, 打开池配置信息。
- (3) 通过 ID\_PROCESSOR 接入 DSP。
- (4) 在 DSP 上装载 DSP 可执行文件 (scale.out)。
- (5) 为数据传输建立 CHNL\_ID\_INPUT 和 CHNL\_ID\_OUTPUT 通道。
- (6) 在这些通道上分配和初始化指定大小的缓冲区。
- (7) 客户端在 DSP 上启动执行。
- (8) 打开远程传输。

### • 执行

(1) 一直查找在 DSP 端产生的 MSGQ。如果定位呼叫不成功 (DSP 序列还没有产生), 会休眠一段时间后再尝试定位序列。

(2) 在 CHNL\_ID\_OUTPUT 上分配缓冲区并一直等待回收缓冲区。

(3) 回收完成表明缓冲区已经通过物理链路传输。

(4) 在 CHNL\_ID\_INPUT 上分配一个空缓冲区, 并一直等待回收缓冲区。

(5) 一旦回收完成, 它的内容会和发布在 CHNL\_ID\_OUTPUT 上的缓冲区比较。DSP 端的应用程序被一个比例因子初始化, 它用来测量数据规模。

(6) 每 100 次的数据传输迭代, 客户端发送一个消息到 DSP 端的 MSGQ, 这个 MSGQ 有一个新的缩放因子。接着, 更多的缓冲区从 DSP 接收缩放的数据。

(7) 客户端重复第 2 步到第 7 步, 重复次数由使用者定义。

### • 结束

- (1) 客户端关闭远程传输。
- (2) 停止 DSP。
- (3) 客户端释放分配给数据传输的缓冲区。
- (4) 删除 CHNL\_ID\_INPUT 和 CHNL\_ID\_OUTPUT 通道。
- (5) 关闭池。
- (6) 从 ID\_PROCESSOR 中分离。
- (7) 关闭本地传输。
- (8) 最后释放 PROC 组件。

## 2. 在 DSP 端

使用带 SIO 和 MSGQ 的 TSK。

### • 初始化

- (1) 在 main() 函数中, 创建客户端任务 tskScale。
- (2) 根据用于数据驱动的物理链路, 配置应用程序所需的池来分配消息和数据缓冲区。
- (3) 为数据传输创建 SIO 通道 (INPUT\_CHANNEL 和 OUTPUT\_CHAN)。
- (4) 为数据传输分配和初始化缓冲区。
- (5) 在本地处理器打开一个通过具体名字确认的消息队列。

### • 执行

- (1) 这个任务是在 INPUT\_CHANNEL 上分配缓冲区并一直等待回收缓冲区。



(2) 这个任务设法在本地队列上得到一个消息。获取操作是没有超时指定。如果它在指定的 MSGQ 已经可以获得, 结果会在返回的消息中。

(3) 如果消息可用, 可以从消息中提取新的缩放因子。缩放因子用于从 GPP 接收多缓冲区内容。

(4) 在 OUTPUT\_CHANNEL 上分配缩放的缓冲区并一直等待回收。

(5) 回收完成表明 GPP 上的客户端已经收到缓冲区。

(6) 这些步骤一直重复, 直到迭代次数作为参数传递给 DSP, 执行才完成。

#### • 结束

(1) 在删除阶段, 任务首先删除本地消息队列。

(2) 然后删除 SIO 通道。

(3) 释放分配给数据传输的缓冲区。

使用带 GIO 和 MSGQ 的 SWI。

#### • 初始化

(1) 从 main() 函数中调用 SWISCALE\_create。

(2) 根据用于数据驱动的物理链路, 配置应用程序所需的池来分配消息和数据缓冲区。

(3) 为数据传输建立两个 GIO 通道 (INPUT - CHANNEL 和 OUTPUT - CHANNEL)。

(4) 建立两个 SWI 对象, 一个用于数据传输 (命名为 dataSWI), 另外一个用于消息传输 (命名为 msgSWI)。当数据通道上的读或写请求完成后, dataSWI 会被自动调用; 当收到一个消息后, msgSWI 会被自动调用。

(5) 分配和初始化用于数据传输的缓冲区。

(6) 在本地处理器 (DSP) 打开一个通过具体名字确认的消息队列。

#### • 执行

(1) 在输入缓冲区启动数据传输的一个读请求, 提交请求到 INPUT\_CHANNEL。

(2) 一旦 SWI 发布, 输入缓冲区的内容根据缩放因子缩放并传输到输出缓冲区。

(3) 空的输入缓冲区被重新发到输入通道, 填充完成的缓冲区被发送到输出通道。

(4) 完成两次请求后, SWI 被再次发布。

(5) 每当在 MSGQ 上收到消息, 消息的 SWI 被发布。SWI 会检查在没有超时指定下, 通过得到一个消息来确定消息是否可用。从这个消息中得到一个新的缩放因子, 并将其保存。从收到缩放因子那时候起, 将其用于所有的数据缓冲区缩放。

(6) 重复第 1 到第 5 步, 直到 GPP 应用程序发布缓冲区。

#### • 结束

在这个例程中, 由于读写请求, 数据 SWI 会连续发布。同样的, 在本地消息队列无论消息是否准备好, 消息 SWI 也连续发布。所以它们永远不会达到完成阶段。终止序列步骤如下。

(1) 删除 SWI 的数据和消息。

(2) 删除本地消息队列。

(3) 删除 GIO 通道 (INPUT\_CHANNEL 和 OUTPUT\_CHANNEL)。

(4) 释放为数据传输分配的缓冲区。

### 3. 启动应用程序

这个 scale 例子带有下面的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	./scale.out
缓冲区大小	1 024
迭代次数	10 000

### 3.4.5.4 READ WRITE

这个例程描述了大缓冲区传输，这个传输是通过从 DSP 存储器直接读写。使用 API 函数 PROC\_Read() 和 PROC\_Write() 在 GPP 和 DSP 之间实现大尺寸的缓冲区传输。在 DSP 端，这个应用程序描述了带 MSGQ 的 TSK 的使用。

示例应用程序中的数据和信息流如图 3.51 所示。

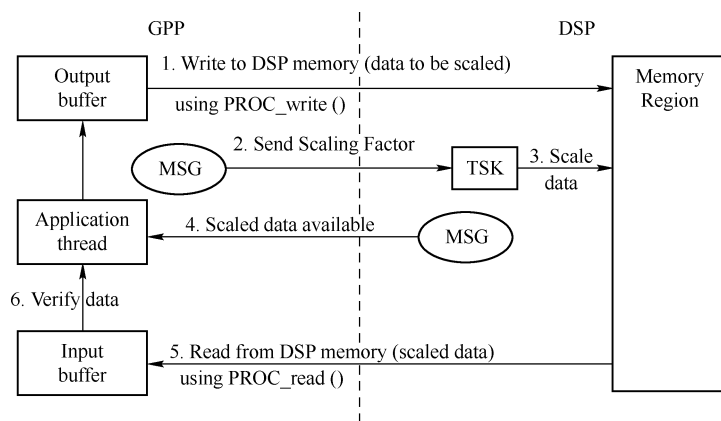


图 3.51 READWRITE 示例中数据和信息流

#### 1. 在 GPP 端

##### • 初始化

- (1) 客户端设置必要的数据结构访问 DSP。
- (2) 通过 ID\_PROCESSOR 接入到 DSP。
- (3) 根据用于数据驱动的物理链路，打开分配消息需要的池。
- (4) 打开消息队列，这个队列通过本地处理器的名字来识别。
- (5) 给上一步的队列设置错误处理程序。
- (6) 在 DSP 上装载可执行文件 (readwrite.out)。
- (7) 客户开始在 DSP 上运行。
- (8) 然后打开远程传输。
- (9) 一直查找 DSP 端打开的队列。如果查找呼叫失败 (DSP 队列没有打开)，会休眠一段时间后再次尝试查找队列。

##### • 执行

- (1) 客户端分配输入输出缓冲区需要的缓冲区。
- (2) 客户端对数据区域赋初值，为确保数据传输正确，允许数据完整性检查。
- (3) 使用 API 函数 PROC\_Write() 对 DSP 写数据缓冲区。

- (4) 客户端写一个消息给 DSP, 通知 DSP 数据缓冲区已经被写在 DSP 上。
- (5) 客户端给 DSP 端发送一个消息, 这个 MSGQ 带有一个新的压缩因子。接着, 从 DSP 接收到更多的缓冲区。
- (6) 从 DSP 等待一个消息, 用于确认写入的数据。
- (7) 使用 API 函数 PROC\_Read(), 客户端从 DSP 区域读。
- (8) 数据完整性检查, 确保从 GPP 写到 DSP 和 GPP 从 DSP 读到的缓冲区内容的有效性。

- 结束

- (1) 客户在 DSP 端释放远程队列。
- (2) 关闭远程传输通道。
- (3) 停止 DSP 执行。
- (4) 重设建立阶段的错误处理程序。
- (5) 关闭本地消息队列。
- (6) 关闭池。
- (7) 从 ID\_PROCESSOR 分离。
- (8) 释放 PROC 组件。

## 2. 在 DSP 端

使用带 MSGQ 的 TSK。

- 初始化

- (1) 通过全局变量 POOL\_config 静态配置池, 用于消息发送。
- (2) 通过全局变量 MSGQ\_config 静态配置 MSGQ 组件。
- (3) 在 main() 函数中建立客户任务 tskReadWrite。
- (4) 在本地处理器上, 通过指定名称打开消息队列。
- (5) 给上一步的队列设置错误处理程序。
- (6) 一直查找 DPP 端打开的队列。如果查找呼叫失败 (DPP 队列没有打开), 会休眠一段时间后再次尝试查找队列。查找操作是同步的。

- 执行

- (1) 本任务设法在本地队列上得到一个消息, 这个消息是通知数据已经被 GPP 写好。这个操作被指定为永久等待。
- (2) 如果消息可用, 一个新缩放因子就会从消息中提取。这个缩放因子用来增加从 GPP 收到的缓冲区的内容, 结果值写在不同的区域。
- (3) 然后发布消息, 说明用于数据传输的缩放缓冲区已经准备好。
- (4) 这些步骤重复进行, 循环次数由用户指定。

- 结束

- (1) 客户端在 DSP 上发布远程消息队列。
- (2) 重设建立阶段的错误处理程序。
- (3) 关闭本地消息队列。

## 3. 启动应用程序

这个 readwrite 例子带有下面的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	./readwrite.out
DSP 地址	2 156 986 368
缓冲区大小	1 024
迭代次数	10 000

3.4.5.5 MAPREGION

这个例程描述了通过 DSP/BIOS LINK，直接指针访问 DSP 存储区。本例程只支持 DM642\_PCI 平台。图 3.52 显示了例程的运行状态。在 DSP 端，本例程与 readwrite 例程一样。

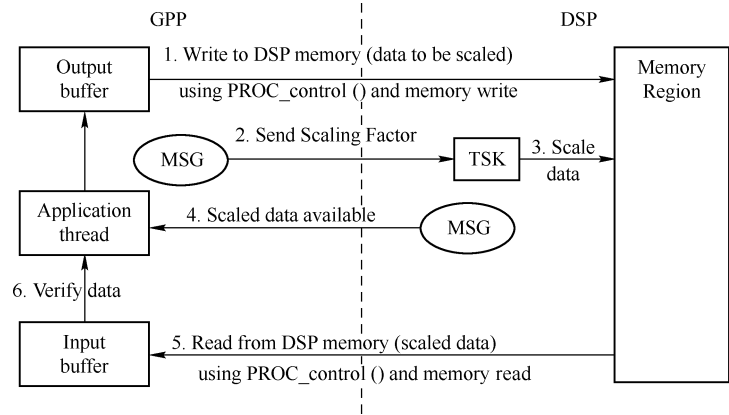


图 3.52 MAPREGION 示例中数据和信息流

1. 在 GPP 端

• 初始化

- (1) 客户端设置必要的数据结构访问 DSP。
- (2) 通过 ID\_PROCESSOR 接入到 DSP。
- (3) 根据用于数据驱动的物理链路，打开分配消息需要的池。
- (4) 打开消息队列，这个队列通过本地处理器的名字来识别。
- (5) 给上一步的队列设置错误处理程序。
- (6) 在 DSP 上装载可执行文件 (readwrite.out)。
- (7) 客户端启动在 DSP 上的执行。
- (8) 然后打开远程传输。
- (9) 查找 DSP 端打开的队列。查找会一直进行。如果查找呼叫失败 (DSP 队列没有打开)，会休眠一段时间后再次尝试查找队列。

• 执行

- (1) 客户端分配输入输出缓冲区需要的缓冲区。
- (2) 客户端对数据区域赋初值，为确保数据传输正确，允许数据完整性检查。
- (3) 使用 API 函数 PROC\_Write() 对 DSP 写数据缓冲区。
- (4) 客户端写一个消息给 DSP，通知 DSP 数据缓冲区已经被写在 DSP 上。

(5) 客户端给 DSP 端发送一个消息，这个 MSGQ 带有一个新的压缩因子。接着，从 DSP 接收到更多的缓冲区。

(6) 从 DSP 等待一个消息，用于确认写入的数据。

(7) 使用 API 函数 PROC\_Read()，客户端从 DSP 区域读。

(8) 数据完整性检查，确保从 GPP 写到 DSP 和 GPP 从 DSP 读到的缓冲区内容的有效性。

• 结束

(1) 客户在 DSP 端释放远程队列。

(2) 关闭远程传输通道。

(3) 停止 DSP 执行。

(4) 重设建立阶段的错误处理程序。

(5) 关闭当地消息队列。

(6) 关闭池。

(7) 从 ID\_PROCESSOR 分离。

(8) 释放 PROC 组件。

2. 在 DSP 端

DSP 的使用与 readwrite 例程一样。

3. 启动应用程序

这个例子带有下面的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	./readwrite. out
缓冲区大小	1 024
迭代次数	10 000

3.4.5.6 RING\_IO

这个例程描述了使用 DSP/BIOS LINK 中的 RingIO 组件，使用两个 RingIO 实例实现 GPP 和 DSP 间的数据流。在 GPP 上运行的任务和 DSP 上运行的任务之间传输数据。

在 DSP 端，这个应用描述了使用带 RingIO 的 TSK。

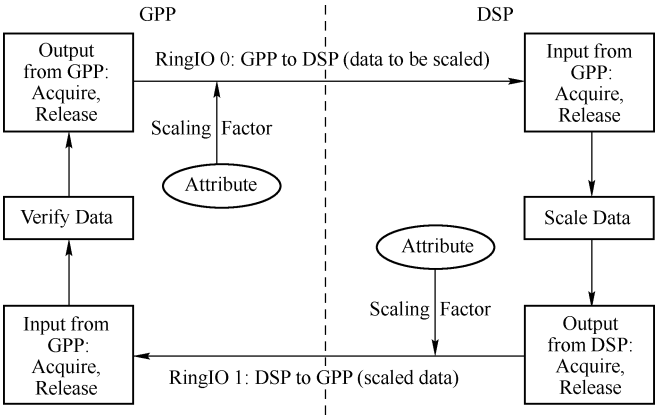


图 3.53 RINGIO 示例中数据流

## 1. 在 GPP 端

### • 初始化

- (1) 为了访问 DSP, 客户端调用 API。
- (2) 为 DSP 初始化 RingIO 组件。
- (3) 打开池, 分配 RingIO 数据缓冲区、属性缓冲区、控制结构和锁定对象。
- (4) 通过 ID\_PROCESSOR 接入到 DSP。
- (5) 在 DSP 上装载执行文件 ringio.out。
- (6) 客户开始在 DSP 上执行。
- (7) 建立 RingIO (RingIO1), 把 DPP 的数据写到 DSP。
- (8) 然后打开写者模式的 RingIO1 (GPP 建立)。
- (9) 一直等待开放式调用 RingIO2 读者模式成功。调用成功, 表明 RingIO 已经被 DSP 建立。

### • 执行

(1) 客户端为数据传输的读写双方设置通知, 以标记特定的缓冲区大小。指向旗语的指针被传到每个通知函数, 由通知函数传递旗语, 以唤醒因为等待旗语而阻塞的应用程序。

(2) 为了写入 RingIO1, GPP 获得缓冲区。

- 获得相应数目的缓冲区, 写入程序中的迭代次数。其模为 MAX\_BUFFERS。
- 如果缓冲区不可用, 应用等待一个信号量。这个信号量是在为读者注册的通知功能里。

(3) 这套缓冲区中, GPP 写入到它们的值是用缩放因子来表示的。缩放因子的初始值为 1, 它的值随着每一次迭代递增, 它的模为 0xFF。

(4) 在获得缓冲区的开始阶段, GPP 设置一个属性值。这个属性值表明了 DSP 缓冲区的缩放因子。

(5) 写完每个缓冲区后, GPP 释放获得的缓冲区。

(6) 这时 DSP 会以读模式从 RingIO1 中获得缓冲区, 拷贝它们的内容到缓冲区, 这些缓冲区是以写模式从 RingIO2 得到的。

(7) GPP 以读模式从 RingIO2 中获得缓冲区, 并基于存在缓冲区的缩放因子来验证数据内容。完成数据验证后, 释放缓冲区。

(8) 根据指定的迭代次数, 重复 2~7 步。

### • 结束

- (1) 客户端关闭以读者模式打开的 RingIO。
- (2) 客户端关闭以写者模式打开的 RingIO。
- (3) 删除 GPP 端建立的 RingIO。
- (4) 停止 DSP 执行。
- (5) 关闭池。
- (6) 结束 DSP 的 RingIO 组件。
- (7) 从 ID\_PROCESSOR 分离。
- (8) 最后释放 PROC 组件。

## 2. 在 DSP 端

### • 初始化



- (1) 在 `main()` 函数中建立客户任务 `tskRingIo`。
- (2) 初始化 `RingIO` 和 `NOTIFY` 组件。
- (3) 建立 `RingIO` (`RINGIO2`) 用于 DSP 写到 GPP 的数据传输。
- (4) 打开 `RINGIO2` (DSP 创建的) 写者模式。
- (5) 一直等到公开调用 `RINGIO2` 写者模式成功, 表明 `RINGIO` 已经被 GPP 创建。
- (6) 为从 GPP 收到的缩放数据分配临时缓冲区。

#### • 执行

(1) 客户端为数据传输的读写双方设置通知, 以标记特定的缓冲区大小。指向旗语的指针被传到每个通知函数, 由通知函数传递旗语, 以唤醒因为等待旗语而阻塞的应用程序。

(2) 任务从 `RingIO1` 读模式中获得完整的缓冲区。拷贝获得的缓冲区内容到临时缓冲区, 并释放它们。

(3) 如果一个属性可用, 设置收到属性的缩放因子。

(4) 然后任务以写模式从 `RingIO2` 中获得缓冲区。当用当前的因子缩放内容后, 拷贝临时缓冲区的内容到获得的缓冲区。

(5) 为每个迭代次数在 `RingIO2` 上设置一个属性, 属性带有某次迭代的缩放因子, 缩放因子用于缓冲区的传输。

(6) 在缩放数据写完以后, 释放从 `RingIO2` 获得的缓冲区。

(7) 根据指定的迭代次数, 重复 2~6 步。

#### • 结束

(1) 在删除阶段, 任务首先释放分配的所有临时缓冲区。

(2) 关闭写者模式中打开的 `RingIO`。

(3) 关闭读者模式中打开的 `RingIO`。

(4) 删除 DSP 建立的 `RingIO`。

(5) 最后完成 `NOTIFY` 组件。

### 3. 启动应用程序

这个 `RingIO` 的例子带有下面的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	<code>./ringio.out</code>
缓冲区大小	128
迭代次数	10 000

#### 3.4.5.7 MP\_LIST

这个例程描述了使用 DSP/BIOS LINK 中的 `MPLIST` 组件, 使用多处理器列表实例实现 GPP 和 DSP 间的数据流。在 GPP 上运行的任务和 DSP 上运行的任务之间传输数据。

在 DSP 端, 这个应用描述了带 `MPLIST` 的 `TSK`。

##### 1. 在 GPP 端

#### • 初始化

(1) 为了访问 DSP, 客户端调用 API。

(2) 为 DSP 初始化 `MPLIST` 组件。

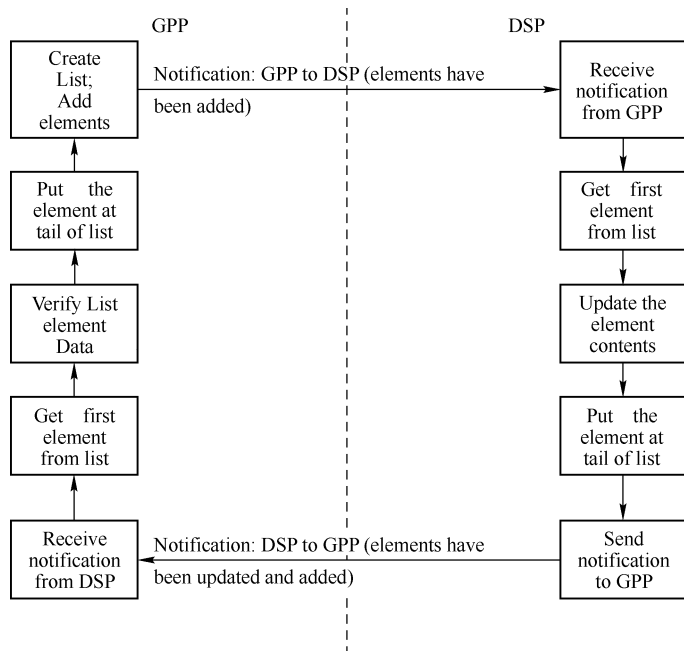


图 3.54 MP\_LIST 例程中数据流

(3) 为分配 MPLIST 的数据结构，打开池。数据结构包括列表本身和由用户指定的元素的数目。

(4) 通过 ID\_PROCESSOR 接入到 DSP。

(5) 在 DSP 上装载执行文件 mplist. out。

(6) 客户开始在 DSP 上执行。

(7) 建立 MPLIST (MPLIST 实例)，把数据写到 DSP。

(8) 客户端开始在 DSP 上执行。

(9) 当 DSP 改变了列表元素，客户端建立通知，使 GPP 知道。

#### • 执行

(1) 客户端为 MPLIST 和用户指定的列表元素分配存储器。

(2) 客户端得到关于创建 MPLIST 的句柄。

(3) GPP 使用迭代次数和它在列表的位置来设立列表元素。然后列表元素被添加到列表的尾部。这样做是因为元素个数是由用户指定的。

(4) 当元素已经被添加到列表，GPP 发送一个消息给 DSP，让 DSP 做同样的改变。

(5) 然后等待 DSP 表明 DSP 已经在共享列表中完成改变元素的通知。

(6) 收到通知后，它表明被 DSP 完成的修改是一个函数的迭代次数和列表的位置。

(7) 根据指定的迭代次数，重复 2~6 步。

#### • 结束

(1) 客户端关闭以读者模式打开的 MPLIST。

(2) 删除 MPLIST 例程。

(3) 停止 DSP 执行。

(4) 关闭池。

(5) 从 ID\_PROCESSOR 分离。

(6) 最后释放 PROC 组件。

## 2. 在 DSP 端

### • 初始化

(1) 在 main() 函数中建立客户任务 tskMpList。

(2) 初始化 MPLIST、NOTIFY 和 MPCS 组件。

(3) 打开 GPP 创建的 MPLIST 实例 (GPPMPLIST)。这时它能够得到一个句柄, 以便能够执行列表操作。

(4) 注册一个事件回调, 当 GPP 已经添加所有列表元素到列表后, DSP 能得到通知。

### • 执行

(1) 任务等待从 GPP 端发来的通知, 通知会表明所有列表元素已经被添加到了列表。

(2) 通知被接收后, DSP 从表中弹出表头。修改列表元素的数据结构: 通过一个函数设置元素的值, 包括元素迭代次数及元素在列表中的位置。然后添加元素在列表尾部。为所有元素进行上述处理。

(3) 完成修改后, DSP 发送一个通知到 GPP, 让 GPP 做同样的改变。

(4) 根据指定的迭代次数, 重复 1~3 步。

### • 结束

(1) 在删除阶段, 关闭 MPLIST 句柄。

(2) 为事件回调关闭注册。

(3) 在删除阶段, 任务首先释放所有分配的临时缓冲区。

## 3. 启动应用程序

这个 MP\_LIST 例子带有下列的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	./mplist.out
迭代次数	10 000
元素个数	100

### 3.4.5.8 MPCSXFER

这个例程描述了通过带有互斥访问保护共享缓冲区的机制, 实现 GPP 和 DSP 之间的数据传输。使用 POOL 组件分配共享缓冲区, 并使用 MPCS 组件对共享缓冲区提供访问保护。使用 NOTIFY 组件实现 GPP 和 DSP 端应用程序的同步。

在 DSP 端, 这个应用程序描述了带 MPCS、POOL 和 NOTIFY 组件的 TSK 的使用。

## 1. 在 GPP 端

### • 初始化

(1) 为接入 DSP, 客户端建立必要的数据结构。

(2) 通过 ID\_PROCESSOR 接入到 DSP。

(3) 为了分配共享的 MPCS 对象、控制和数据缓冲区, 需要打开池。

(4) 分配控制和数据缓冲区, 翻译它们的地址到 DSP 的地址空间, 并发送到 DSP。

(5) 建立和打开 MPCS 对象, 用来保护控制和数据缓冲区。

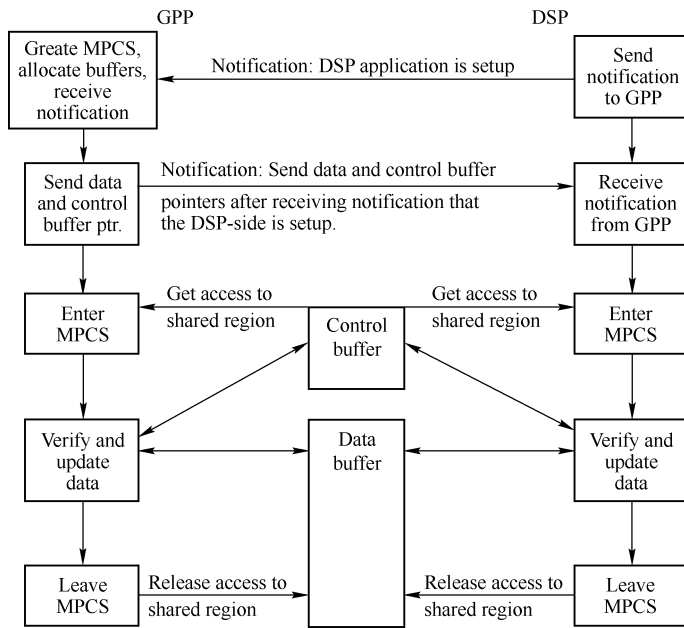


图 3.55 MCPSXFER 例程中的数据流

(6) 初始化控制缓冲区内容, 通过 POOL 同步缓冲区内容后写回缓冲区。

(7) 创建一个旗语, 用来等待从 DSP 发来的通知, 并为应用程序的相关事件注册该通知。

(8) 在 DSP 上装载可执行文件 (mpcsxfer.out)。

(9) 客户开始在 DSP 上执行。

(10) 然后等待旗语。当旗语被传递, 表明 DSP 应用程序已经完成设置。

(11) 给 DSP 发送事件, 带有该事件的控制和数据缓冲区的 DSP 地址。

#### • 执行

(1) 通过使用提供互斥访问的 MPCS 进入缓冲区, 客户端试图获得共享控制和数据缓冲区。

(2) 对于控制和数据缓冲区内容, 跨处理器同步是无效的。

(3) 如果控制缓冲区的内容表明 DSP 已经更新了控制和数据缓冲区, 它们的内容会被验证。在这种情况下, 如果控制缓冲区指示为空, 控制和数据缓冲区的内容被修改, 表明 GPP 已经更新了它们。

(4) 如果控制缓冲区内容表明 GPP 是最后更新的, 客户端会休眠几微秒, 以便模拟完成其他的处理。

(5) 将控制和数据缓冲区的内容写回, 以实现跨处理器同步内容。

(6) 通过离开 MPCS, 客户端释放控制缓冲区。

(7) 根据用户指定的次数或者无限次, 重复 1~6 步。

#### • 结束

(1) 客户端停止 DSP 执行。

(2) 从 DSP 取消事件的注册通知, 删除等待消息的旗语。

(3) 关闭 MPCS 对象的句柄并删除它。

- (4) 释放为控制和数据缓冲区分配的 pool 存储器。
- (5) 关闭池。
- (6) 从 ID\_PROCESSOR 分离。
- (7) 释放 PROC 组件。

## 2. 在 DSP 端

### • 初始化

(1) 被用于数据传输的池，其使用 MPCS 组件。它通过全局变量 POOL\_config 静态配置。

- (2) 在 main() 函数中建立客户端任务 tskMpcsXfer。
- (3) 在本地处理器上使用指定名称打开 MPCS 对象。
- (4) 注册被应用程序使用的事件的通知。
- (5) 发送一个事件通知到 GPP，表明已经完成设置。

(6) 等待从 GPP 端来的事件回调，发出旗语表明收到控制缓冲区指针。第二个事件回调表明收到数据缓冲区指针。

### • 执行

(1) 通过使用提供互斥访问的 MPCS 进入缓冲区，客户端试图获得共享控制和数据缓冲区。

(2) 对于控制和数据缓冲区内容，跨处理器同步是无效的。

(3) 如果控制缓冲区的内容表明 DPP 已经更新了控制和数据缓冲区，它们的内容会被验证。在这种情况下，如果控制缓冲区指示为空，控制和数据缓冲区的内容被修改，表明 DSP 已经更新了它们。

(4) 如果控制缓冲区内容表明 DSP 是最后更新的，客户端会休眠几微秒，以便模拟完成其他的处理。

(5) 将控制和数据缓冲区的内容写回，以实现跨处理器同步内容。

(6) 通过离开 MPCS，客户端释放控制缓冲区。

(7) 根据用户指定的次数或者无限次，重复 1~6 步。

### • 结束

(1) 客户端从 GPP 取消事件的注册。

(2) 对 MPCS 对象关闭句柄。

## 3. 启动应用程序

这个 MPCSXFER 例子带有下列的参数。

参 数	参 数 值
DSP 可执行文件的绝对路径	./mpcsxfer.out
缓冲区大小	128

## 参 考 文 献

- [1] David Dart. DSP/BIOS II Technical Overview. Texas Instruments Incorporated, March 2000.

- 
- [2] Texas Instruments. TMS320 DSP/BIOS v5.41 User Guide. Texas Instruments Incorporated, August 2009.
  - [3] Texas Instruments. DSP/BIOS Textual Configuration (Tconf) User's Guide. Texas Instruments Incorporated, May 2006.
  - [4] Texas Instruments. TI SYS/BIOS v6.33 Real-time Operating System User's Guide. Texas Instruments Incorporated, December 2011.
  - [5] Texas Instruments. TMS320C6000 DSP/BIOS5.x API Reference Guide. Texas Instruments Incorporated, November 2010.
  - [6] Texas Instruments. TMS320C55x DSP/BIOS 5.x API Reference Guide. Texas Instruments Incorporated, August 2009.
  - [7] Don S. Gillespie. Programming and Debugging Tips for DSP/BIOS. Texas Instruments Incorporated, May 2001.
  - [8] David Dart. DSP/BIOS Kernel Technical Overview. Texas Instruments Incorporated, August 2001.
  - [9] Prashanth L A, Eldo Tony Kuruvilla. Synchronizing DSP/BIOS Threads. Texas Instruments Incorporated, May 2004.
  - [10] Harish Thampi S. DSP/BIOS Timers and Benchmarking Tips. Texas Instruments Incorporated, July 2002.
  - [11] Eric Wilbur, Rafael de Souza, Mohsen Khayami. Getting Started With the C6000 Network Development Kit (NDK) . Texas Instruments Incorporated, July 2008.
  - [12] Texas Instruments. TI Network Developer's Kit (NDK) v2.24 API. Texas Instruments Incorporated, July 2014.
  - [13] Texas Instruments. TMS320C6000 Network Developer's Kit (NDK) Software User's Guide. Texas Instruments Incorporated, January 2009.
  - [14] Texas Instruments. TMS320C6000 Network Developer's Kit (NDK) v 2.00 Software Programmer's Reference Guide. Texas Instruments Incorporated, January 2009.
  - [15] Texas Instruments. DSP/BIOS Driver Developer's Guide DDK version 1.20. Texas Instruments Incorporated, August 2005.
  - [16] Don S. Gillespie. Using DSP/BIOS I/O in Multichannel Systems. Texas Instruments Incorporated, October 2000.
  - [17] Texas Instruments. The TMS320DM642 Video Port Mini-Driver. Texas Instruments Incorporated, August 2003.
  - [18] Texas Instruments. DSP/BIOS LINK User Guide Version 1.40.03. Texas Instruments Incorporated, October 2006.



## 第4章 优化的DSP库

DSP 库包括很多通用的信号处理、数学运算以及图像和视频处理等子程序。这些库里所包含的子程序经过汇编优化，可由 C 语言调用。DSP 库里的程序专门用于计算性的增强型实时程序中，这些程序对运行的最佳速度要求非常严格。使用 DSP 库里的子程序，开发者可以获得比用标准 C 语言编写的程序更快的速度。TI 公司提供了一系列库函数帮助开发者来完成各种复杂的算术运算。这样既减少了程序员的工作量，又提高了程序效率。在程序开发中合理使用这些库函数将大大提高系统性能。本章对 TI 公司 DSP 的算法库、数学库和图像处理库等进行详细的介绍。

### 4.1 DSP 的算法库 DSPLIB

DSPLIB 包含了自适应滤波、相关、快速傅里叶变换、卷积、矩阵计算等许多常用的数字信号处理算法，而这些算法正是数字信号处理的基石。

DSPLIB 的核心是一系列经过手工优化的汇编程序代码，这些代码封装在后缀名为 .lib 的文件中，完成各种运算，它们对外是不可见的。这些程序（库函数，routines）可被 C 程序调用，由于经过了手工优化，它们的效率都非常高。另外，不同系列 DSP 芯片的指令集不同。因此，不同系列 DSP 芯片的 DSPLIB 是不同的，如 TMS320C5000 的 DSPLIB 就不能用于 TMS320C6000。但是，各个系列 DSPLIB 的基本组成是相同的，一个完整的 DSPLIB 通常由以下几个部分组成。

- lib 文件夹：存放 \*.lib 文件，内部封装手工优化的汇编程序代码，是 DSPLIB 的核心部分。有的 DSPLIB 还有 \*.src 文件，这些 \*.src 文件主要是用 C 语言和汇编语言编写的程序源代码，使用归档器可从中提取出这些源代码。
- include 文件夹：用于存放各个库函数的头文件，通常这些文件分为 C 程序头文件和汇编程序头文件两部分。
- 其他辅助文件。

#### 4.1.1 DSPLIB 的下载和安装

由于 DSPLIB 种类繁多，且属可选模块，通常的 DSP 开发环境（CCS，Code Composer Studio）并没有配备 DSPLIB。因此，使用一个 DSPLIB 之前，必须进行 DSPLIB 的下载和安装。

下载：在 TI 公司网站 [www.ti.com](http://www.ti.com) 上可免费下载各种 DSPLIB。

安装：DSPLIB 下载完毕后，双击安装文件，将它安装在计算机中选定的位置（默认位置为 C:\ti）。

安装之后，就可以在程序开发中使用 DSPLIB 的库函数。

程序员并非只能被动地使用 DSPLIB，只要遵循相应的规则，程序员就可以自己编写一个 DSPLIB，编写一个最简单的 DSPLIB 的步骤如下。

(1) 新建一个工程 newLibrary，将其属性设为“Library (.lib)”，如图 4.1 所示。



图 4.1 编写自己的 DSPLIB

(2) 编写高效率代码文件 myLib1.asm、myLib2.asm、myLib3.asm，…。

(3) 将 myLib1.asm、myLib2.asm、myLib3.asm，…添加到工程 new Library 中。

(4) 编译链接工程 new Library。

完成上面 4 步后，工程中就出现库文件 newLibrary.lib，一个 DSPLIB 就制作成功了。为了使 DSPLIB 具有保密性，通常情况下只保留工程中的 newLibrary.lib 文件，而将其他文件，特别是源代码文件 \*.asm 删除或保密存放。这样，用户就只能使用库文件，而无法从中得到源代码的信息。

#### 4.1.2 利用 DSPLIB 实现 FFT 算法

离散傅氏变换 (DFT, Discrete Fourier Transform) 广泛应用于离散信号的数字信号处理，它完成离散信号时域到频域的转换。直接的 DFT 运算需要约  $N^2$  次乘法操作， $N$  为采样点个数。

20 世纪 60 年代由 Cooley 和 Turkey 提出的快速傅氏变换 (FFT, Fast Fourier Transform) 是 DFT 中用到的快速算法中的一种，广泛应用于许多实际的应用中，包括快速 FIR 滤波器、频谱分析和合成、以及互相关计算等。FFT 可以明显地降低运算量，只需要约  $(N/2 \cdot \log_2 N)$  次乘法，它已经成为数字信号处理的基本工具和迅速发展的动力。FFT 也成为评价数字器件与系统性能的标准之一。随着 FFT 的广泛应用，研究人员做了大量的工作来改善其性能。一方面是算法的改进，另一方面是硬件的实现。算法研究人员已经研究了一系列在不增加存储资源的条件下提高 FFT 运算速度的算法，而 VLSI 系统开发者也在不断改善系统的性能，为 FFT 的应用提供方便。尤为突出的是单片可编程 DSP 器件，不论现在已经使用的，还是正在开发的，都能实现高速 FFT 运算，使得由它组成的复杂系统的实时处理能力大为提高。

以下介绍了分别利用 C54x DSPLIB、C55x DSPLIB、C62x DSPLIB、C64x DSPLIB、C67x DSPLIB 实现的标准 FFT 算法。

### 4.1.2.1 利用 C54x DSPLIB 实现标准的 FFT 算法

TI 公司提供了专门支持 C54x 的 DSPLIB，其核心库文件为 C54xDSP.lib。该库中包含了一组全面的、优化了的 FFT 和 IFFT 汇编程序，这些程序如表 4.1 所示。它既支持 16-bit 的复数和实数 DIT 基 2-FFT/IFFT 运算，如 cfft、cfft、rfft 和 rfft 等函数；又支持 32-bit 的复数 DIT 基 2-FFT/IFFT 运算，如 cfft32 和 cfft32 等函数。同时，该库还包含了位倒序程序 cbrev，该程序重排输入的复数序列，使其满足位倒序的顺序。输入的复数序列在调用 cfft 或 cfft32 函数前，需要调用 cbrev 程序进行位倒序。同样，输入的实数序列在调用 rfft 程序前，也需要调用 cbrev 程序。对于  $N$  点的实数输入序列，rfft 程序等同于实现一个大小为  $N/2$  的 cfft。利用一个  $N/2$  点复 FFT 算法实现一个  $N$  点实值序列的 FFT，rfft 程序可以节省相同的运算量。

表 4.1 TMS320C54x DSP 库中的  $N$  点 FFT/IFFT 程序

FFT/IFFT function	Description	Cycle count	Code size, text section	Data size, sintab section
cfft	Complex radix-2 DIT FFT with optional scale factor $1/N$	8 542 [ $N=256$ ] 19 049 [ $N=512$ ] 42 098 [ $N=1\,024$ ]	343 [ $N=256$ ] 391 [ $N=512$ ] 439 [ $N=1\,024$ ]	367 [ $N=256$ ] 750 [ $N=512$ ] 1 517 [ $N=1\,024$ ]
cfft	Complex radix-2 DIT IFFT with optional scale factor $1/N$	Same as cfft	Same as cfft	Same as cfft
cfft32	32-bit complex radix-2 DIT IFFT (similar to cfft)	30 059 [ $N=256$ ] 144 205 [ $N=512$ ] 371 312 [ $N=1\,024$ ]	498 [ $N=256$ ] 521 [ $N=512$ ] 544 [ $N=1\,024$ ]	764 [ $N=256$ ] 1,514 [ $N=512$ ] 3 050 [ $N=1\,024$ ]
cfft32	32-bit complex radix-2 DIT IFFT (similar to cfft)	29 446 [ $N=256$ ] 142 724 [ $N=512$ ] 361 469 [ $N=1\,024$ ]	498 [ $N=256$ ] 521 [ $N=512$ ] 544 [ $N=1\,024$ ]	764 [ $N=256$ ] 1,514 [ $N=512$ ] 3 050 [ $N=1\,024$ ]
rfft	Real radix-2 DIT FFT (in place) with optional Scale factor $1/N$	5 470 [ $N=256$ ] 11 881 [ $N=512$ ] 25 716 [ $N=1\,024$ ]	357 [ $N=256$ ] 405 [ $N=512$ ] 453 [ $N=1\,024$ ]	367 [ $N=256$ ] 750 [ $N=512$ ] 1 517 [ $N=1\,024$ ]
rfft	Real IFFT (in place)	Same as rfft	Same as rfft	Same as rfft
cbrev	Obtain bit-reversal order array	$3N+23$ (off place), $13N-5$ (in place)	50	$2N$ (off place) $N$ (in place)

在表 4.1 中，对于复数 FFT 程序， $N$  就等于输入序列的长度；对于实数 FFT 程序， $N$  则为输入序列长度的一半。在表中，通过把 cbrev 程序和 cfft（或者 rfft）运行的周期数相加，再除以处理器的时钟频率，就可以得到执行一个复数（或者实数）FFT 所需要的时间。例如，对于一个 1 024 点的复数 FFT 运算，在一个 100MHz 的 C54x 处理器上执行 cfft 程序，其运行时间大约为 0.55 毫秒。与之相对，对于一个 1 024 点的实数 FFT 运算，程序 rfft 的运行时间大约只需要 0.32 毫秒。同时，和精度为 16-bit 的程序相比，精度为 32-bit 的程序的执行时间明显增多。

表 4.1 还提供了各个库函数的代码长度和使用存储空间的大小。观察表 4.1 可得，和精度为 16-bit 的复 FFT/IFFT 程序相比，精度为 32-bit 的复 FFT/IFFT 程序的代码长度明显增加。对于处理 16-bit 数据的复数和实数 FFT/IFFT 程序来说，它们所需要的存储空间是一样的。如果处理数据变为 32-bit，则存储空间的使用量为 16-bit 的两倍。对于位倒序程序 cbrev，同址计算会减小所需存储空间的大小，即从  $2N$  降为  $N$ 。但是，其代价是运行周期由

3N 提高到 13N。

另外，还可以控制是否对表 4.1 中的 FFT/IFFT 程序进行缩放。当 SCALE 选项设为 0 时，不执行缩放；否则，缩放因子设置为  $N$ ，即在 FFT 的每一级按比例 0.5 进行缩放。采用逐渐缩放的原因主要是基于精度损失的考虑。例如，如果一个 256 点的 IFFT 在输入时缩小到  $1/256$ ，立即就会损失 8-bit 的精度，极大地降低了输入数据的精度。一个更好的办法就是，在每一级进行 0.5 的逐渐缩放。这样，既可以保证  $1/N$  的缩放，又能维持一个较好的精度。

#### 4.1.2.2 利用 C55x DSPLIB 实现标准 FFT 算法

与 C54x 的 DSPLIB 相似，C55x 的 DSPLIB 同样提供了一系列的 FFT/IFFT 函数和位倒序程序，用于实数和复数的 FFT 计算，其函数列表如表 4.2 所示。在表 4.2 中还给出了这些函数的执行时间（周期数）和代码的大小。和 C54x 的 DSPLIB 相比，C55x DSPLIB 的复数 FFT 和 IFFT 程序的执行时间更短，代码量更小。C55x DSPLIB 中的实数 FFT 函数，即 rfft，实际上是在 dsplib.h 中定义的一个宏，它调用 cfft 函数和 cbrev 函数，其调用方式如下。

表 4.2 TMS320C55x 的 DSP 库中 FFT/IFFT 函数表

FFT/IFFT function	Description	Cycle count	Code size .text section
cfft	Complex radix-2 DIT FFT with optional scale factor; in-place computation; required twiddle. inc, which contains the twiddle factors	With scaling: 6 183 [ $N=256$ ] 13 638 [ $N=512$ ] 30 043 [ $N=1\,024$ ] Without scaling: 5 563 [ $N=256$ ] 12 434 [ $N=512$ ] 27 689 [ $N=1\,024$ ]	With scaling: 490 Without scaling: 361
cifft	Radix-2 DIT IFFT with optional scale factor	Very close to cfft	Very close to cfft
rfft	Radix-2 real DIT FFT, equivalent to $N/2$ complex FFT (cfft)		
rifft	Radix-2 real DIT FFT, equivalent to $N/2$ complex IFFT (cifft)		
cbrev	Bit reversal for both complex and real FFT	$2N$ (off place) $4N+6$ (in place)	81

```
#define rft(x,nx,type)\
{\
    cfft_##type (x, nx/2), \
    cbrev (x, x, nx/2), \
    unpack (x, nx)\
}
```

#### 4.1.2.3 利用 C62x DSPLIB 实现标准 FFT 算法

表 4.3 给出了 C62x DSPLIB 中提供的复数 FFT 和 IFFT 函数。在 C62x 实现 FFT 过程中存在一个潜在问题，那就是存储器组的冲突。因此，信号数组和旋转因子系数数组应该在不同的数据段或存储器空间。在 C62x DSPLIB 中，没有同址的 FFT/IFFT 函数，所有的相关函数

都是异址的，如表 4.3 所示。

表 4.3 C62x DSPLIB 的 FFT 函数表

FFT/IFFT function	Description	Cycle count	Code size
DSP_bitrev_cplx	Complex bit – reversal routine	$7(N/4 + 2) + 18$	352
DSP_radix2	Complex radix – 2 DIF FFT	$\log_2 N \times (4N/2 + 7) + 34 + N/4$	800
DSP_r4fft	Complex radix – 4 DIF FFT; input may be scaled by $1/N$	$\log_4 N \times (10 \times N/4 + 29) + 36 + N/4$	736
DSP_fft16 × 16r	Complex mixed radix FFT; scaled by 0.5 at each stage	$2.5N \times \text{ceil}[\log_4 N] - N/2 + 164$	1 344

#### 4.1.2.4 利用 C64x DSPLIB 实现标准 FFT 算法

C64x 的 DSPLIB 同样提供了一套 FFT/IFFT 程序。并且，C64x 向下兼容 C62x 的 DSPLIB。另外，在 C64x DSPLIB 中，还提供了一些附加函数，以充分利用 C64x 的额外精度。表 4.4 给出了 C64x DSPLIB 中的 FFT 相关函数及其代码长度和代码执行时间。和 C62x 的 DSPLIB 相比，C64x 的库函数的执行速度要快 2 ~ 3 倍。

表 4.4 C64x DSPLIB 的 FFT 函数列表

FFT/IFFT Func.	Description	Cycle Count	Code size (bytes)
DSP_bitrev_cplx	Backwards compatible with C62x DSPLIB		
DSP_radix2	Backwards compatible with C62x DSPLIB		
DSP_r4fft	Backwards compatible with C62x DSPLIB		
DSP_fft	Complex FFT (radix4), off place	$1.25\log_4 N - 0.5N + 23\log_4 N - 1$	984
DSP_fft16x16r	Complex mixed radix $16 \times 16$ – bit FFT with scaling, off place	$\text{ceil}(\log_4 N - 1) \times (1.25N + 25) + 0.25N + 26$	868
DSP_fft16x16t	Complex mixed radix $16 \times 16$ – bit FFT with scaling, off place	$(1.25N + 19) \times \text{ceil}(\log_4 N - 1) + 1.5(N + 8) + 27$	1 004
DSP_fft16x32	Complex mixed radix $16 \times 32$ – bit FFT, off place	$(13N/8 + 24) \times \text{ceil}(\log_4 N - 1) + 1.5(N + 8) + 27$	1 068
DSP_fft32x32	Complex mixed radix $32 \times 32$ – bit FFT, off place	$(2.5N + 20) \times \text{ceil}(\log_4 N - 1) + 1.5N + 39$	932
DSP_fft32x32s	Complex mixed radix $32 \times 32$ – bit FFT with scaling, off place	$(2.5N + 20) \times \text{ceil}(\log_4 N - 1) + 1.5N + 39$	932
DSP_ifft16x32	Complex mixed radix $16 \times 32$ – bit IFFT, off place	$(13N/8 + 25) \times \text{ceil}(\log_4 N - 1) + 1.5(N + 8) + 30$	1 064
DSP_ifft32x32	Complex mixed radix $32 \times 32$ – bit IFFT, off place	$[10(N/4 + 1) + 10] \times \text{ceil}(\log_4 N - 1) + 6(N/4 + 2) + 27$	932

#### 4.1.2.5 利用 C67x DSPLIB 实现标准 FFT 算法

C67x 的 DSPLIB 同样提供了一套 FFT/IFFT 程序，用以支持各种浮点 FFT 计算。表 4.5 给出了 C67x DSPLIB 中的 FFT 相关函数及其代码长度和代码执行时间。

表 4.5 C67x DSPLIB 的 FFT 函数列表

Function	Description	Cycle Count	Code size (bytes)
DSPF_sp_bitrev_cplx	Complex bit reverse	$5N/2 + 26$	608
DSPF_sp_cfft4_dif	Complex radix 4 FFT Using DIF	$(7N/2 + 23) \log_4 N + 20$	1 184
DSPF_sp_cfft2_dit	Complex radix 2 FFT using DIT	$(2N \log_2 N) + 42$	1 248
DSPF_sp_fftSPxSP	Cache optimized mixed radix FFT	$3N \times \text{ceil}(\log_4 N - 1) + 44 + 21 \text{vceil}(\log_4 N - 1) + 2N$	1 440
DSPF_sp_iftSPxSP	Cache optimized mixed radix inverse FFT with complex input	$3N \times \text{ceil}(\log_4 N - 1) + 44 + 21 \times \text{ceil}(\log_4 N - 1) + 2N$	1 472
DSPF_sp_icefft2_dif	Complex radix 2 inverse FFT	$2N \log_2 N + 37$	1 600

### 4.1.3 利用 DSPLIB 实现无限单位冲激响应 (IIR) 数字滤波器

数字滤波器是数字信号处理最重要的组成部分之一，在各个领域得到了广泛的应用。所谓数字滤波，就是将输入的信号序列，按规定的算法处理后，得到所希望的输出序列的过程。因此，一个数字滤波器可以看作是一个数字系统，它的特性可以用系统传输函数，或者说系统的单位冲激响应来表示。那么，按照单位冲激响应到底是无限的还是有限的，可以将数字滤波器分为无限冲激响应 (IIR, Infinite Impulse Response) 数字滤波器和有限冲激响应 (FIR, Finite Impulse Response) 数字滤波器两种。

模拟滤波器的性能，在结构确定之后，主要取决于器件的宽容度。与此类似，数字滤波器的性能，在结构确定之后，主要取决于算法的精度。特别是 IIR 滤波器对溢出十分敏感，在用定点 DSP 来实现时，累加器 ACC 的有限宽度可能产生溢出。因而，需要对其中间节点进行分析，了解不同形式的 IIR 滤波器为了防止溢出而对定标和移位所提出的不同要求。这是在 DSP 上实现 IIR 滤波器需要重点研究的部分。

IIR 滤波器是数字信号处理最基本的组成单元之一。和 FIR 滤波器相比，达到相同的性能，IIR 滤波器需要的阶数或系数更少。这就意味着在滤波过程中，它需要更少的存储单元，是一种高效的滤波器。目前，IIR 滤波器得到了广泛的应用，并且已经成为评价 DSP 性能的一个重要标准。

IIR 滤波器主要有以下几个特点。

#### ➤ 非线性相位

IIR 滤波器在频带范围内的相位响应并不是线性的。因此，群时延在不同的相位点是不同的，从而会产生相位失真。

#### ➤ 稳定性问题

由于 IIR 滤波器的反馈特性，它有可能是不稳定的。因此，在设计 IIR 滤波器需要十分小心以保证其极点都在单位圆内，从而使滤波器稳定。这一点在使用定点设备实现时需要特别加以注意。

#### ➤ 有限精度误差

IIR 对有限字长效应特别敏感。因此，在设计时常常使用二阶节级联的结构来减少累加误差的影响。



### ➤ 运算量

和 FIR 滤波器相比, IIR 滤波器可以使用较低的阶数来达到高阶 FIR 滤波器的性能。因此, IIR 所需要的计算量, 主要是指乘法和加法次数, 较 FIR 滤波器而言要小得多。但是, 在 DSP 处理器上实现 IIR 滤波器比较复杂, 需要更长的代码长度来完成对前馈和反馈部分的操作。

### ➤ 陡峭的频率响应

相对于 FIR 滤波器, IIR 滤波器的频率响应更加陡峭。它的极点对于陡峭的频率响应有着极大的贡献。

综上所述, 和 FIR 滤波器相比, 在设计和实现 IIR 滤波器时需要非常细心。目前已经有了许多比较成熟的 IIR 滤波器设计方法和软件工具。其中较为常用的一种就是将传统的经典模拟滤波器转换为等效的数字 IIR 滤波器。

以下介绍了分别利用 C2x DSPLIB、C54x DSPLIB、C55x DSPLIB、C62x DSPLIB、C64x DSPLIB、C67x DSPLIB 实现标准的 IIR 数字滤波器。

#### 4.1.3.1 利用 C2x DSPLIB 实现标准 IIR 数字滤波器

C2x 同样提供了 DSPLIB 来完成滤波运算。例如, C24x 就提供了一个专门的 Filter Library, 里面包含了各种滤波程序。为了实现 IIR 滤波, 它提供了两个汇编子程序, IIR5B1Q16 和 IIR5B1Q32, 可以供基于 C 语言或汇编语言的主程序调用。第一个子程序处理 16-bit 的定点数据, 第二个子程序处理 32-bit 的数据。

#### 4.1.3.2 利用 C54x DSPLIB 实现标准 IIR 数字滤波器

C54x DSPLIB 提供了一组汇编优化的 IIR 函数, 包括单数据处理和数据块处理, 如表 4.6 所示。这些函数使用的数据格式为 Q.15 或是 Q.31 格式。其中, 函数 iir32 使用的是 32-bit 的系数和数据, 这意味着滤波器的系数和中间变量的数据格式都是双精度的。与单精度 IIR 滤波器相比, 其代价是函数 iir32 需要 12 倍的时钟周期以及两倍的代码长度。另外 3 个函数 (iircas4、iircas5 和 iircas51) 则使用二阶节级联的方式完成 IIR 滤波。4 系数函数 (iircas4) 中指定前馈部分的第一个系数  $b_0 = 1$ , 虽然少用了一个系数, 但是其适应性不如 5 系数的函数。

同时, 这些函数都可以给出溢出的状态, 但是并没有避免溢出的程序段。因此, 需要程序员自己选用合适的缩放方法以保证不会产生溢出。例如, 程序员需要在调用直接二型 IIR 函数之前, 对输入信号进行缩放因子为  $G_1$  的缩放。

另外, 尽管函数的输入信号仍然是 Q.15 格式, 但是, 滤波器系数必须使用 QN.M 格式, 以保证能够表示最大的系数。在这个格式中, 参数  $N$  表明系数的整数部分由  $N-1$  位表示。

表 4.6 C54x DSPLIB 中实现 IIR 滤波器的函数,  $N_k$  为二阶节的数量,  $N_x$  为输入长度

IIR Func.	Description	Cycles	Code size (words)
iir32	5 Coefficients Cascade IIR Filter Direct Form II, 32-bit	$N_x(12 + 56N_k) + 62$	110
iircas4	4 Coefficients Cascade IIR Filter Direct Form II	$N_x(11 + 5N_k) + 40$	50
iircas5	5 Coefficients Cascade IIR Filter Direct Form II	$N_x(11 + 6N_k) + 40$	51
iircas51	5 Coefficients Cascade IIR Filter Direct Form I	$N_x(13 + 8N_k) + 44$	58

#### 4.1.3.3 利用 C55x DSPLIB 实现标准 IIR 数字滤波器

和 C54x 类似, C55x DSPLIB 也给出了一组汇编优化的 IIR 函数, 以供 C 程序调用, 如表 4.7 所示。函数 iir32 使用的是 32-bit 的系数和数据, 即双精度格式。函数 (iircas4、iircas5 和 iircas51) 则使用单精度 16-bit 格式的二阶节级联的方式完成 IIR 滤波。其中, 4 系数函数 iircas4 中指定前馈部分的第一个系数  $b_0 = 1$ , 虽然少用了一个系数, 但是其适应性不如 5 系数的函数 iircas5。最后一个函数 iircas51 使用的是  $K$  个直接 I 型的二阶节级联来实现 IIR 滤波。正如讨论的一样, 这种实现 IIR 滤波器的方式需要的存储空间和运算时间都较高。因此, 最适合的 IIR 函数应该是 iircas5。

表 4.7 C55x DSPLIB 中实现 IIR 滤波器的函数,  $Nk$  为二阶节的数量,  $Nx$  为输入长度

IIR Func.	Description	Cycles	Code size (bytes)
iir32	5 Coefficients Cascade IIR Filter Direct Form II, 32-bit	$Nx(7 + 31Nk) + 77$	203
iircas4	4 Coefficients Cascade IIR Filter Direct Form II	$Nx(2 + 3Nk) + 44$	122
iircas5	5 Coefficients Cascade IIR Filter Direct Form II	$Nx(5 + 5Nk) + 60$	126
iircas51	5 Coefficients Cascade IIR Filter Direct Form I	$Nx(5 + 8Nk) + 68$	154

#### 4.1.3.4 利用 C62x DSPLIB 实现标准 IIR 数字滤波器

在 C62x 的 DSPLIB 中提供了两个实现 IIR 滤波器的函数。一个是 DSP\_iir, 用于实现二阶节级联的 IIR 滤波器, 另一个为 DSP\_iirlat, 用于实现全极点的 IIR 格型滤波器, 它们的其他信息如表 4.8 所示。

表 4.8 C62x DSPLIB 中实现 IIR 滤波器的函数,  $Nk$  为系数长度,  $Nx/N$  为输入/出长度

IIR Func.	Description	Cycles	Code size (words)
DSP_iir	5 Coeff Cascade Biquad IIR Filter	$5N + 30$	384
DSP_iirlat	All-pole IIR Lattice IIR Filter	$(2Nk + 8) \times Nx + 5$	352

#### 4.1.3.5 利用 C64x DSPLIB 实现标准 IIR 数字滤波器

C64x 的 DSPLIB 同样提供了实现 IIR 滤波器的函数, 函数名和 C62x 的相同。它们的信息如表 4.9 所示。

表 4.9 C64x DSPLIB 中实现 IIR 滤波的函数,  $Nk$  为系数长度,  $Nx/N$  为输入/出长度

IIR Func.	Description	Cycles	Code size (words)
DSP_iir	5 Coeff Cascade Biquad IIR Filter	$4N + 21$	276
DSP_iirlat	All-pole IIR Lattice IIR Filter	$(2Nk + 7) \times Nx + 9$	352

#### 4.1.3.6 利用 C67x DSPLIB 实现标准 IIR 数字滤波器

在 C67x 的 DSPLIB 中提供了三个函数来完成和 IIR 滤波器相关的运算, 它们是 DSPF\_sp\_biquad、DSPF\_sp\_iir 和 DSPF\_sp\_iirlat, 分别用于实现二阶节级联结构的 IIR 滤波器、VSELP 编码中的 IIR 滤波器 (used in VSELP vocoder) 和全极点的格型 IIR 滤波器。它们的信息如

表 4.10 所示。

**表 4.10 C67x DSPLIB 中实现 IIR 滤波的函数， $N_k$  为系数长度， $N_x/N$  为输入/出长度**

IIR Func.	Description	Cycles	Code size (bytes)
DSPF_sp_biquad	5 Coeff Cascade Biquad	$4N_x + 76$	1 312
DSPF_sp_iir	used in VSELP vocoder	$6N + 59$	1 152
DSPF_sp_iirlat	All - pole IIR Lattice IIR Filter	$\lceil 6\text{floor}(N_k + 1)/4 + 29 \rceil N_x$	1 024

#### 4.1.4 利用 DSPLIB 实现有限单位冲激响应 (FIR) 数字滤波器

IIR 滤波器的优点是需要的阶数少、计算复杂程度低。并且，在设计 IIR 数字滤波器时，可以利用模拟滤波器设计的结果。但是，它也有明显的缺点。首先是 IIR 滤波器存在稳定性问题。其次是相位的非线性，若需线性相位，则需要采用全通网络进行相位校正。这是一个严重的问题，因为很多应用，如图像处理以及数据传输等，都要求信道具有线性相位特性。而 FIR 滤波器就可以做成具有严格的线性相位，同时又可以具有任意的幅度特性。此外，FIR 滤波器的单位冲激响应是有限长的，因而滤波器一定是稳定的。再有，只要经过一定的时延，任何非因果有限长序列都能变成因果的有限长序列，因而总能用因果系统来实现。最后，FIR 滤波器由于单位冲激响应是有限长的，因此可以用快速傅立叶变换 FFT 算法来实现过滤信号，从而可大大提高运算效率。FIR 滤波器的缺点就是要取得很好的频谱衰减特性，其单位冲激响应的长度（或阶数）要比 IIR 滤波器长得多。

因此，在滤波器的设计过程中，如果与幅度响应相比，相位失真的重要性是第二位的，则所设计的数字滤波器可以由需要较少存储空间、计算复杂程度较低、从而成本也比较低的 IIR 结构来实现。然而，如果在通带中要求线性相位及特定的幅度响应，则往往使用 FIR 结构来实现。

FIR 滤波器主要有下面几个优点。

- 线性相位：FIR 滤波器的响应可以是严格线性相位的，这样，在整个频带范围内的群时延是一个常数。因此，滤波器本身不会引入相位失真。
- 绝对稳定：由于没有反馈，FIR 滤波器始终都是稳定的。
- 较小的有限精度误差：FIR 滤波器受有限字长效应，如系数量化误差和舍入误差的影响都较小。
- 实现效率高：利用 DSP 处理器的 MAC（乘 - 累加单元）、循环寻址以及一些专用的指令可以使 FIR 滤波器的实现比较高效。

但是在相同的滤波性能下，和 IIR 滤波器相比，FIR 滤波器需要更高的阶数，或更多的系数，即需要更长的延时，更大的运算代价，更多的储存器空间。此外，对于 FIR 滤波器而言，没有与之相对应的模拟滤波器，因此计算和优化 FIR 滤波器都需要计算机程序进行大量的运算。

一般而言，FIR 滤波器的相位和群时延特性优于 IIR 滤波器。所以，在应用中如果对波形的要求较高，FIR 滤波器是比较好的选择。如果需要窄带、锐截止的幅 - 频响应特性，而且相位不是非常重要的时候，IIR 滤波器优于 FIR。

以下分别介绍利用 C2x DSPLIB、C54x DSPLIB、C55x DSPLIB、C62x DSPLIB、C64x DSPLIB、C67x DSPLIB 实现的 FIR 数字滤波器。

#### 4.1.4.1 利用 C2x DSPLIB 实现标准 FIR 数字滤波器

在 C24x 的 DSPLIB 中，也提供了一系列实现 FIR 滤波器的函数。表 4.11 中列出了这些函数，它们都可以在 C 程序中直接调用。

在调用这些 FIR 函数时，需要一个延迟缓冲区（delay buffer）来储存以前的输入信号。这个缓冲区必须是在 C2x 的内部 DARAM 上的，它可以通过 DMOV 指令进行操作。如果要使用循环缓冲区，那么数据在缓存区中必须以适当的方式对齐（appropriate buffer alignment）。

表 4.11 C24x DSPLIB 中的 FIR 滤波函数

FIR Func.	Description
firfilt_gen	Generic FIR filter, using linear buffer
firfilt_ord10	10 – order FIR filter, using linear buffer
firfilt_ord20	20 – order FIR filter, using linear buffer
firfilt_cgen	Generic FIR filter, using circle buffer
firfilt_cord10	10 – order FIR filter, using circle buffer
firfilt_cord20	20 – order FIR filter, using circle buffer

C24x 的 DSPLIB 提供了通用（generic）和固定阶数（fixed – order）形式的 FIR 函数。通用的 FIR 函数是使用循环指令来完成滤波的，而固定阶数的 FIR 函数则解开（unroll）了这些循环。因此，通用的 FIR 函数所需要的执行时间更长。

#### 4.1.4.2 利用 C54x DSPLIB 实现标准 FIR 数字滤波器

C54x 的 DSPLIB 提供了一系列用于完成 FIR 滤波器的函数，它们可以在 C 程序中直接调用。表 4.12 中列出这些实现 FIR 滤波的函数，同时给出了各个函数所需要的执行周期以及代码长度（16 – bit word）。表中的最后一项为系统设置和初始化的开销。

从表 4.12 中可知，和实现直接型 FIR 滤波器的函数 fir 相比，实现对称的线性相位 FIR 滤波器的函数 firs2 在运行时间和代码长度上都没有优势。这是因为，虽然和直接型 FIR 滤波器相比，线性相位 FIR 滤波器的系数对称性，使得所需要的乘法次数降低了一半，但是，在实现线性相位 FIR 滤波器时，需要大量的指针操作（例如，它需要两个指向数据的指针）。这样，乘法次数上的优势就被复杂的、额外的指针操作抵消了。

表 4.12 C54x DSPLIB 中实现 FIR 的函数，Nx 和 Nh 分别为输入数据和系数的长度

FIR Func.	Description	Cycles	Code size (16 – bit words)
fir	Direct form FIR filter	$4 + N_x(4 + N_h) + 34$	42
cfir	Complex direct form FIR filter	$N_x(13 + 8N_h) + 49$	66
firdec	Decimating FIR filter with decimation factor D	$(N_x/D) [12 + N_h + 4(D - 1)] + 86$	67
firinterp	Interpolating FIR filter with interpolation factor I	$N_x[6 + (I - 1)(17 + N_h/I)] + 88$	74
firs	Symmetric FIR filter, coefficients in program memory	$N_x(16 + N_h/2) + 35$	56
firs2	Symmetric FIR filter, coefficients may not in program memory	$N_x(15 + N_h) + 43$	58

C54x DSPLIB 中还有一个实现线性相位 FIR 滤波器的函数 `firs`。和 `firs2` 不同，它要求滤波器的系数必须存放在程序空间内，这样就可以得到比较高的运行速度。而且，由于其所需要的乘法次数较少，运算的功耗也相应地较小。但是，它较之直接型 FIR 滤波器的优势也仅仅在滤波器系数大于 18 的时候才能体现。线性相位 FIR 滤波器的另一个优势就是它可以节省一半的系数存储空间。

下面以实现直接型 FIR 滤波器的函数 `fir` 为例，说明其调用方法。

```
oflag = short fir( DATA * x, DATA * h, DATA * r, DATA * * dbuffer, ushort nh, ushort nx)
```

其中，`x[nx]` 为输入的长度为 `nx` 的实数数组；`h[nh]` 为滤波器的系数数组， $h = b_0, b_1, b_2, \dots$ 。注意，它在存储器中需要按  $K$  对齐， $K = \log_2 nh$ ，即 `h` 的起始地址的低  $K$  位必须为 0。`r[nx]` 为输出的长度为 `nx` 的实数数组，允许  $x = r$ ；`dbuffer[nh]` 为延时缓冲区；`nx` 是输入的数据点数；`nh` 为系数的个数；`oflag` 是溢出标志，如果为 1 则表示中间运算或是结果出现了溢出。

C54x 的 DSPLIB 同时还提供了支持抽取（Decimation，每  $D$  样值丢弃  $D - 1$  个）和插值（Interpolation，每两个样值间插入  $I - 1$  个）功能的 FIR 滤波器函数。详细的说明参见表 4.11。

#### 4.1.4.3 利用 C55x DSPLIB 实现标准 FIR 数字滤波器

同样，C55x 的 DSPLIB 提供了一系列用于完成 FIR 滤波器的函数，它们可以在 C 程序中直接调用。表 4.13 中列出这些实现 FIR 滤波的函数，同时给出了各个函数所需要的执行周期以及代码长度。其中， $Nh$  为滤波器系数的长度， $Nx$  表示输入数据的长度， $Nx = 1$  为单点计算， $Nx \geq 2$  为连续计算。表中所有的代码长度以字节为单位，而运行周期的最后一项是系统设置和初始化的开销。

表 4.13 C55x DSPLIB 中实现 FIR 的函数，其中  $Nx$  和  $Nh$  分别为输入数据和系数长度

FIR Func.	Description	Cycles	Code size (bytes)
<code>fir</code>	Direct form FIR filter	$Nx(2 + Nh) + 25$	107
<code>fir2</code>	Block FIR filter	$Nx/2[9 + (Nh - 2)] + 32$	134
<code>firs</code>	Symmetric direct form FIR filter	$Nx(5 + Nh/2) + 72$	133
<code>firs2</code>	Dual - MAC direct form FIR filter	$(Nx/2) \times (9 + Nh - 2) + 32$	134
<code>cfir</code>	Complex direct form FIR filter	$Nx[8 + 2(Nh - 2)] + 51$	136
<code>firdec</code>	Decimating FIR filter, with $D$ the decimation factor	$(Nx/D) \times [10 + Nh + (D - 1)] + 67$	144
<code>firinterp</code>	Interpolating FIR filter, with $I$ the Interpolation factor	$Nx[2 + I \times (1 + Nh/I)] + 72$	164
<code>firlat</code>	Lattice forward FIR filter	$Nx[4 + 4(Nh - 1)] + 23$	53

#### 4.1.4.4 利用 C62x DSPLIB 实现标准 FIR 数字滤波器

C62x 的 DSPLIB 提供了一系列实现 FIR 滤波器的函数，可供 C 程序直接调用。在 C 程序中使用这些函数，既可以节省开发时间，也可以使代码执行效率更高。表 4.14 中列出了



C62x DSPLIB 中实现 FIR 滤波的函数，其中  $N_x$  为处理数据的长度， $N_h$  为系数长度。除了 DSP\_fir\_sym，即实现对称的线性相位 FIR 滤波器的函数，其他函数的数据长度均为 16 - bit，累加器大小均为 32 - bit。而 DSP\_fir\_sym 处理的数据长度为 16 - bit，累加器大小为 40 - bit。另外，不同的函数对系数的长度以及输出点数的要求有所不同。例如，DSP\_fir\_r4 要求滤波器的系数是 4 的倍数且不少于 8，而 DSP\_fir\_r8 则要求滤波器的系数是 8 的倍数且不少于 8。其中，基 4 和基 8 表示 FIR 滤波函数的内部滤波循环分别被展开 4 次和 8 次。

**表 4.14 C62x DSPLIB 中的实现 FIR 的函数， $N_x$  和  $N_h$  分别为输入数据和系数长度**

FIR Func.	Description	Cycles	Code size (bytes)
DSP_fir_gen	General purpose FIR filter	$[9 + 4 \times \text{ceil}(N_h/4)] \text{ceil}(N_x/2) + 18$	640
DSP_fir_r4	FIR filter(radix 4)	$(8 + N_h)N_x/2 + 14$	544
DSP_fir_r8	FIR filter(radix 8)	$N_h N_x/2 + 28$	544
DSP_fir_sym	Symmetric FIR filter	$(3N_h/2 + 10)N_x/2 + 20$	416
DSP_fir_cplx	Complex FIR filter(radix 2)	$2N_h N_x + 20$	384

表中的代码长度都是以字节为单位的，另外，运行周期的最后一项是系统设置和初始化的开销。注意，需要大量的指令来完成复杂的指针操作，对称型 FIR 滤波器并不比其他的实现方式更为高效。

#### 4.1.4.5 利用 C64x DSPLIB 实现标准 FIR 数字滤波器

C64x 的 DSPLIB 中也提供了一系列用于实现 FIR 滤波器的函数，可供 C 程序直接调用。表 4.15 中列出了 C64x DSPLIB 中实现 FIR 滤波的函数，其中  $N_x$  为处理数据的长度， $N_h$  为系数长度。

**表 4.15 C64x DSPLIB 中实现 FIR 的函数， $N_x$  和  $N_h$  分别为输入数据和系数长度**

FIR Func.	Description	Cycles	Code size (bytes)
DSP_fir_gen	General purpose FIR filter	$[11 + 4\text{ceil}(N_h/4)] \text{ceil}(N_x/4) + 18$	640
DSP_fir_r4	FIR filter(radix 4)	$(8 + N_h)N_x/4 + 9$	544
DSP_fir_r8	FIR filter(radix 8)	$N_h N_x/4 + 17$	544
DSP_fir_sym	Symmetric FIR filter	$(10N_h/8 + 15)N_x/4 + 26$	416
DSP_fir_cplx	Complex FIR filter(radix 2)	$N_h N_x + 24$	384

表中的代码长度都是以字节为单位的，另外，运行周期的最后一项是系统设置和初始化的开销。由于 C64x DSPLIB 中的函数利用了 C64x 增强的硬件结构以及指令，因而和 C62x DSPLIB 中对应的函数相比，需要更少的运行周期数。

#### 4.1.4.6 利用 C67x DSPLIB 实现标准 FIR 数字滤波器

在 C67x 的 DSPLIB 中提供了四个实现 FIR 滤波器的函数，供 C 程序直接调用。表 4.16 中列出了它们的基本信息，其中  $N_x$  为处理数据的长度， $N_h$  为系数长度。表中的代码长度都是以字节为单位的，另外，运行周期的最后一项是系统设置和初始化的开销。



表 4.16 C67x DSPLIB 中实现 FIR 的函数,  $N_x$  和  $N_h$  分别为输入数据和系数长度

FIR Func.	Description	Cycles	Code size (bytes)
DSPF_sp_fir_gen	Generic FIR filter	$\{4\text{floor}[(N_h - 1)/2] + 14\} \text{ceil}(N_x/4) + 8$	640
DSPF_sp_fir_r2	FIR filter(radix 2)	$N_h N_x / 2 + 34$	960
DSPF_sp_fir_circ	FIR filter with circularly addressed input	$(2N_h + 10) N_x / 4 + 18$	512
DSPF_sp_fir_cplx	Complex FIR filter	$2N_h N_x + 33$	640

### 4.1.5 利用 DSPLIB 实现自适应滤波器

常规滤波器或固定滤波器具有固定的特性,对于输入信号,根据这个滤波器特性产生相应的输出。也就是,先有了滤波器构成的权系数,然后决定相应的输出值。但是,有的实际应用往往是反过来要求的,即对滤波器输出的要求是明确的,而滤波器特性却无法预先知道。例如,在长话系统中,回波抵消器的理想输出是无回波信号,这个要求是明确的,而系统本身却不能一开始就确定下来,因为它取决于长话系统话路传输条件的变化。像这样的应用就必须依赖自适应滤波技术(adaptive filtering),或者说依赖自适应滤波器。

所谓自适应滤波器,就是其权系数可以根据一种自适应算法来不断修改,使系统的冲激响应能满足给定的性能判断。由于自适应滤波器在未知或时变系统中的明显优势,它在众多领域中得到了广泛的应用,例如用于语音编码的自适应预测器、用于求解逆系统的信道的均衡器、用于噪声控制系统的回波抵消器、噪声消除器等。这些应用很多需要实时处理,并且,绝大多数的应用是基于最新发展的 DSP 来设计的。基于 DSP 的自适应滤波器,比用硬件实现的自适应滤波器有很多优点,其功耗以及体积更小、更容易实现。特别重要的是,修改程序使系统很容易升级,功能进一步完善。

早期的自适应滤波器的研究,主要是针对自适应天线系统和数字传输系统的均衡器来设计的。绝大多数对自适应滤波器的研究是基于 Widrow 提出的最小均方算法(LMS, Least Mean Square)。因为, LMS 算法的设计和实现都较为简单,因而在很多应用场合都非常适用。本章所讨论的自适应滤波器结构是 FIR 类型滤波器结构,逼近算法多采用 LMS 算法。这些结构和技术是自适应技术的基础。当然,现在有越来越多的特定应用,它们需要特定的滤波器结构和自适应算法,以适当的复杂性换取系统性能的提高。但本质还是一样的。有些评估自适应滤波器性能的交互式软件包就把采用 LMS 算法和 FIR 结构的自适应滤波器作为评估的基准。

一个自适应滤波器实现的复杂性,通常用它所需的乘法次数和它的阶数(或存储要求)来衡量。基于 DSP 实现的自适应滤波器系统,其 DSP 的数据吞吐量和数据处理能力也就成了考虑的重要因素。大多数 DSP 都有并行的硬件乘法器、流水结构以及快速的片内存储器,这些资源使自适应滤波器的实现更容易,而且也更为有效。

#### 4.1.5.1 利用 C54x DSPLIB 实现 LMS 算法

C54x 的 DSPLIB 提供了三个用于自适应滤波的函数,这些函数本质上都是利用 LMS 指令来实现自适应滤波的,它们可供 C 代码直接调用。表 4.17 列出了这三个函数及其他

们的基本的特性。其中,  $L$  表示 FIR 滤波器的阶数,  $N_x$  表示输入数据的长度, 表中的代码长度是以 16-bit 的字为单位的, 运行时间的最后一项表示系统设置和初始化所引入的额外开销。

值得注意的是, 这三种算法都是基于延迟 LMS 算法的, 其中, 归一化延迟 LMS 算法 NDLMS (Normalized - Delayed LMS) 的函数 `ndlms` 和归一化延迟块 LMS 算法 NBDLMS (Normalized Block - Delayed LMS) 的函数 `nblms` 都采用归一化的步长因子以提高收敛的速度。NBDLMS 算法把滤波器的  $L$  个系数分成  $nb$  块, 每块的系数有  $bs$  个系数, 即  $L = nb \times sb$ , 每一次迭代仅更新其中的一块系数。

表 4.17 C54x DSPLIB 中用于实现自适应滤波的函数

LMS Func.	Algorithm	Cycles	Code size (16-bit words)
<code>dlms</code>	Delayed LMS	$N \times (14 + 2L) + 45$	62
<code>ndlms</code>	Normalized - Delayed LMS	$N \times [63 + 2(L - 1)] + 52$	144
<code>nblms</code>	Normalized Block - delayed LMS	$N \times [85 + bs + L + (18 + bs)nb] + 88$	144

以 `dlms` 函数为例, 在 C 程序中调用 `dlms` 的形式如下:

```
short oflag = dlms( DATA * x, DATA * h, DATA * r, DATA * * d,
                   DATA * des, DATA step, ushort nh, ushort nx)
```

其中, 数据类型 DATA 与 int 等价,  $x$  是指向输入信号向量的指针;  $h$  是指向滤波器权向量的指针。注意, 滤波器权系数是反向存储的, 即按顺序  $h[n-1]$ ,  $h[n-2]$ ,  $\dots$ ,  $h[0]$  的顺序存储,  $h[n-1]$  存储在地址最低的单元中。并且, 由于滤波器系数采用循环寻址, 所以需要设置系数存储区为循环缓冲区, 即地址单元的低  $k = \log_2 nh$  位必须为零。 $r$  是指向输出向量的指针;  $d$  是指向延迟缓冲区的指针, 并且也必须设置为循环缓冲区;  $des$  是指向滤波器期望信号向量的指针;  $step$  是控制算法收敛速度的步长因子, 值为  $2\mu$ ;  $nh$  为滤波器系数的个数, 滤波器的阶数为  $nh-1$ , 且  $nh \geq 3$ ;  $nx$  为输入和输出数据的个数。`oflag` 是算法溢出标志, 为 1 则表示发生溢出, 为 0 则表示没有溢出。

#### 4.1.5.2 利用 C55x DSPLIB 实现 LMS 算法

与 C54x 类似, C55x 的 DSPLIB 中也提供实现延迟 LMS 算法的函数 `dlms` 和 `dlmsfast`, 它们可供 C 程序直接调用。其中, 函数 `dlms` 实现延迟 LMS 算法, 它使用 LMS 汇编指令。函数 `dlmsfast` 用于延迟 LMS 算法的快速实现。与 `dlms` 不同, 它不使用 LMS 汇编指令, 而是把系数更新和滤波分开操作, 从而可以优化得到一个更好的循环。在 `dlmsfast` 中, 使用了 C55x 的双 MAC 单元, 因此, 输入的信号是成对进行处理的。相应地, 每次有两个系数被更新。`dlmsfast` 特别适用于具有大量采样数据的自适应滤波应用。

另外, `dlms` 要求滤波器的输入数据和系数都必须至少在两个以上, 而 `dlmsfast` 则要求滤波器的系数至少 10 个以上。它们的基本信息如表 4.18 所示。其中,  $L$  表示 FIR 滤波器的阶数,  $N_x$  表示输入数据的长度, 表中的代码长度是以字节为单位的, 运行时间的最后一项表示系统设置和初始化所引入的额外开销。

表 4.18 C55x DSPLIB 中用于实现自适应滤波的函数

LMS Func.	Algorithm	Cycles	Code size (byte)
dlms	Delayed LMS	$N \times [7 + 2(L - 1)] + 26$	122
dlmsfast	Delayed LMS, fast implemented	$N \times /2[26 + 3L] + 71$	322

例如，在 C 程序调用 dlms 的形式如下：

```
ushort oflag = dlms ( DATA * x, DATA * h, DATA * r, DATA * des,
                     DATA * dbuffer, DATA step, ushort nh, ushort nx)
```

从上述调用函数的形式可以看出，其中的参数和 C54x 的 DSPLIB 中的 dlms 函数的参数几乎完全一样，参数的定义几乎也全部一样。只是有一点需要注意，那就是 C55x 的延迟循环缓冲区 dbuffer 是包含一个指针寄存器和长度为 nh + 1 的循环缓冲区。而 C54x 延迟循环缓冲区是一个长度为 nh 的循环缓冲区。

在 C 程序中调用 DSPLIB 中的函数时一定要注意函数的说明。例如，C54x 和 C55x 都是定点 DSP 处理器，所以在 dlms 函数中利用的是 Q.15 格式的 DATA 即 int 型的数据类型，函数中用到的系数、输入数据、滤波器输出数据、期望信号和步长因子都是 DATA 型的数据类型。在上述函数中，头文件 test.h 包含了以数组形式存储的滤波器输入信号和期望信号，还有滤波器的阶数、输入数据的个数、延迟缓冲区的定义等。所以在程序中并没有对这些量进行定义。注意，这个头文件要用户自己定义。

#### 4.1.5.3 利用 C62x、C64x 和 C67x DSPLIB 实现 LMS 算法

C62x 的 DSPLIB 中只有一个用于实现自适应滤波器的函数 DSP\_firlms2，它是基于延迟 LMS 算法的。该函数中使用的误差信号由上一时刻的输入信号计算而得，并且要求滤波器的系数必须是 2 的整数倍，代码的长度是 256 字节，需要的时钟周期是  $3L/2 + 26$ 。

与 C62x 的 DSPLIB 类似，C64x 的 DSPLIB 也提供了一个用于实现自适应滤波器的函数 DSP\_firlms2，它同样是基于延迟 LMS 算法的。该函数的输入信号向量、系数向量和误差信号都是 16-bit 的，最后的滤波器输出是 32-bit 的。需要注意的是，滤波器系数的个数  $L$  必须是 4 的整数倍。整个函数代码的长度是 148 字节，执行周期数是  $3L/4 + 17$ 。

C67x 的 DSPLIB 也提供了一个函数 DSPF\_sp\_fir，它用于实现基于 LMS 算法的自适应 FIR 滤波器，函数名中的“sp”表示它是基于单精度浮点数据格式的。该函数要求滤波器的阶数  $L$  必须大于或等于 6，代码的长度是 1376 字节，需要的时钟周期是  $(L + 35) \times N_x + 21$ ，其中其中  $N_x$  为输入数据的长度。

## 4.2 DSP 的数学库 MATHLIB

德州仪器 (TI) 数学库是优化的浮点数学函数库，用于使用 TI 浮点器件的 C 编程器。这些例程通常用于计算密集型实时应用，最佳执行速度是这些应用的关键。通过使用这些例程，而不是在现有运行时支持库中找到的例程，使用者可以在无须重写现有代码的情况下获得更快的执行速度。MATHLIB 库包括目前在现有实时支持库中提供的所有浮点数学例程。这些新函数可称为当前实时支持库名称或包含在数学库中的新名称。

MATHLIB 具有以下特性。

- 自然 C 源码。
- 优化的 C 代码。
- 手工编码、经汇编语言优化的例程。
- C 调用的例程，可内联且与 TMS320C6000 编译器完全兼容。
- 接受单样片或向量输入的例程。
- 提供的函数经 C 模型和现有实时支持函数测试。
- 基准（周期和代码大小）。
- 使用代码生成工具 v7.2.0 进行编译。

MATHLIB 所有函数被分成两类：单精度和双精度浮点数。双精度函数的所有输入参数和输出参数都为双精度数，同时单精度函数的所有输入参数和输出参数都为单精度数。MATHLIB 函数代码都同时支持大、小写的形式。下面将详细介绍 MATHLIB 所包含的函数。

### 4.2.1 三角函数

MATHLIB 提供的三角函数有反正切函数、余弦函数和正弦函数三种。

反正切函数有单操作数和双操作数两种形式。单操作数反正切函数返回一个浮点参数  $a$  的反正切值，返回值角度范围在  $[-\pi/2, \pi/2]$  之间。

- 双精度函数形式为：double atandp(double a)  
说明：如果  $|a| < 1.49\text{e}-8$ ，就直接返回  $a/b$  的值。
- 单精度函数形式为：float atansp(float a)  
说明：如果  $|a| < 2.44\text{e}-4$ ，就直接返回  $a/b$  的值。

双操作数反正切函数返回一个浮点参数  $a/b$  的反正切值，返回值角度范围在  $[-\pi/2, \pi/2]$  之间。

- 双精度函数形式为：double atan2dp(double a, double b)  
说明：如果  $|a/b| < 1.49\text{e}-8$ ，就直接返回  $a/b$  的值。  
如果  $a$  或  $b$  不是一个数，可能返回一个随机数。
- 单精度函数形式为：float atan2sp(float a, float b)  
说明：如果  $|a/b| < 2.44\text{e}-4$ ，就直接返回  $a/b$  的值。  
如果  $a$  或  $b$  不是一个数，可能返回一个随机数。

余弦函数返回一个弧度为  $a$  的余弦值，返回值范围在  $[-1.0, +1.0]$  之间。

- 双精度函数形式为：double cosdp(double a)  
说明：如果  $|a| > 1.0737\text{e}+9$ ，返回值为零。
- 单精度函数形式为：float cossp(float a)  
说明：如果  $|a| > 1.04858\text{e}+6$ ，返回值为零。

正弦函数返回一个弧度为  $a$  的正弦值，返回值范围在  $[-1.0, +1.0]$  之间。

- 双精度函数形式为：double cosdp(double a)  
说明：如果  $|a| < 9.536743\text{e}-7$ ，返回值为 W。  
如果  $|a| > 1.0737\text{e}+9$ ，返回值为零。
- 单精度函数形式为：float cossp(float a)

说明：如果  $|a| < 2.44e-4$ ，返回值为  $a$ 。

如果  $|a| > 1.04858e+6$ ，返回值为零。

### 4.2.2 除法函数和倒数函数

除法函数返回一个  $a$  除以  $b$  的浮点数。

- 双精度函数形式为：double divdp(double a, double b)

说明：如果  $|b| < 2.225e-308$ ，返回为非数值。

- 单精度函数形式为：float divsp(float a, float b)

说明：如果  $|b| < 1.1755e-38$ ，返回为非数值。

倒数函数是求浮点数  $a$  的倒数，返回一个浮点数。

- 双精度函数形式为：double recipdp(double a)

说明：如果  $a$  为非数值，返回值为随机数。

如果  $a$  为零，返回一个非数值。

如果  $|a| > 1.797693e+308 = 2^{1024}$ ，返回值为零。

- 单精度函数形式为：float recipsp(float a)

说明：如果  $a$  为非数值，返回值为随机数。

如果  $a$  为 0.0，返回一个非数值。

如果  $|a| > 3.402823e+38 = 2^{128}$ ，返回值为零。

### 4.2.3 平方根函数和平方根倒数函数

平方根函数是求浮点数  $a$  的平方根，返回一个浮点数。

- 双精度函数形式为：double sqrt dp(double a)

说明：如果  $|a| \leq 0$ ，对于 C、内联和矢量其返回为 0；对于汇编其返回值为  $\text{rsqrt}(|a|)$ 。

- 单精度函数形式为：float sqrtsp(float a)

说明：如果  $|a| \leq 0$ ，对于 C、内联和矢量其返回为 0.0；对于汇编其返回值为非数值或  $\text{sqrt}(|a|)$ 。

平方根倒数函数是求浮点数  $a$  平方根的倒数，返回一个浮点数。

- 双精度函数形式为：double rsqrt dp(double a)

说明：如果  $a \leq 0$ ，对 C、内联和矢量返回值为 8.99E307，对汇编语言返回值为  $\text{rsqrt}(|a|)$ 。

- 单精度函数形式为：float rsqrtsp(float a)

说明：如果  $|a| \leq 0$ ，对于 C、内联和矢量其返回为无穷大；对于汇编其返回值为非数值或  $\text{rsqrt}(|a|)$ 。

### 4.2.4 指数函数

指数函数是求一个数的指数，其底数可以分别为 10、2 或  $e$ 。

以 10 为底的指数函数，返回一个浮点数。

➤ 双精度函数形式为：double exp10dp(double a)

说明：如果  $|a| < 3.34e-17$ ，返回值为 1.0。

如果  $a < -307.6527$ ，返回值为 0.0。

如果  $a > +308.2547$ ，返回值为  $1.797693e+308 = 2^{1024}$ （就是最大的双精度浮点数）。

➤ 单精度函数形式为：float exp10sp(float a)

说明：如果  $a < -37.9298$ ，返回值为 0.0。

如果  $a < -256$ ，对汇编语言返回值可能为随机数。

如果  $a > +38.5318$ ，返回值为  $3.402823e+38 = 2^{128}$ （就是最大的单精度浮点数）。

以 2 为底的指数函数，返回一个浮点数。

➤ 双精度函数形式为：double exp2dp(double a)

说明：如果  $|a| < 1.1e-16$ ，返回值为 1.0。

如果  $a < -1022$ ，返回值为 0.0。

如果  $a > +1024$ ，返回值为  $1.797693e+308 = 2^{1024}$ （就是最大的双精度浮点数）。

➤ 单精度函数形式为：float exp2sp(float a)

说明：如果  $a < -126$ ，返回值为 0.0。

如果  $a < -256$ ，对汇编语言返回值可能为随机数。

如果  $a > +128$ ，返回值为  $3.402823e+38 = 2^{128}$ （就是最大的单精度浮点数）。

以 e 为底的指数函数，返回一个浮点数。

➤ 双精度函数形式为：double expdp(double a)

说明：如果  $|a| < 7.69e-17$ ，返回值为 1.0。

如果  $a < -708.3964$ ，返回值为 0.0。

如果  $a > +709.7827$ ，返回值为  $1.797693e+308 = 2^{1024}$ （就是最大的双精度浮点数）。

➤ 单精度函数形式为：float expsp(float a)

说明：如果  $a < -87.3365$ ，返回值为 0.0。

如果  $a < -256$ ，对汇编语言返回值可能为随机数。

如果  $a > +88.7228$ ，返回值为  $3.402823e+38 = 2^{128}$ （就是最大的单精度浮点数）。

#### 4.2.5 对数函数

对数函数是求一个数的对数，其底数可以分别为 10、2 或 e。

以 10 为底的对数函数，返回一个浮点数。

➤ 双精度函数形式为：double log10dp(double a)

说明：如果 a 为非数值，返回值为随机数。

如果  $a \leq 0$ ，对于汇编其返回值为  $-7.8E307$ ；对于 C、内联和矢量其返回为  $-1.8E308$ 。

➤ 单精度函数形式为：float log10sp(float a)

说明：如果 a 是无穷大，对于汇编其返回值为 38.65221；对于 C、内联和矢量其返



回为 308.254 7。

以 2 为底的对数函数，返回一个浮点数。

➤ 双精度函数形式为：double log2dp(double a)

说明：如果  $a$  为非数值，返回值为随机数。

如果  $a \leq 0$ ，对于汇编其返回值为负无穷大；对于 C、内联和矢量其返回为  $-1.8E308$ 。

➤ 单精度函数形式为：float log2sp(float a)

说明：如果  $a$  是无穷大，对于汇编其返回值为 128.399 8；对于 C、内联和矢量其返回为 1 024。

以  $e$  为底的对数函数，返回一个浮点数。

➤ 双精度函数形式为：double logdp(double a)

说明：如果  $a$  为非数值，返回值为随机数。

如果  $a \leq 0$ ，对于汇编其返回值为负无穷大。

➤ 单精度函数形式为：float logsp(float a)

说明：如果  $a$  是无穷大，对于汇编其返回值为 88.999 99；对于 C、内联和矢量其返回为 709.782 7。

#### 4.2.6 幂指数函数

幂指数函数返回一个以  $a$  为底  $b$  为幂数的数，即  $a^b$ 。

➤ 双精度函数形式为：double powdp(double a, double b)

说明：如果  $a$  或  $b$  为非数值，返回值为随机数。

如果  $a < 0$ ，同时  $b$  不是整数值，则返回值为非数值。

如果  $a = 0$ ，同时  $b < 0$ ，对于汇编其返回值为 0，对于 C、内联和矢量其返回为无穷大。

如果  $|a|$  为无穷大， $b < 0$ ，对于汇编其返回值为无穷大，对于 C、内联和矢量其返回为 0。

➤ 单精度函数形式为：float powsp(float a, float b)

说明：如果  $a$  或  $b$  为非数值，返回值为随机数。

如果  $b = 0$ ，返回值为 1.0 ( $a$  值被忽略)。

如果  $a < 0$ ，同时  $b$  不是整数值，则返回值为非数值。

如果  $a = 0$ ，同时  $b < 0$ ，对于汇编其返回值为 0，对于 C、内联和矢量其返回为无穷大。

如果  $|a|$  为无穷大， $b < 0$ ，对于汇编其返回值为无穷大，对 C、内联、向量返回值为 0。

如果  $a = b = 1.175\,494\,351E - 38$ ，对于汇编其返回值为 0，对 C、内联、向量返回值为 1。

## 4.3 DSP 的 IQmath 数学函数库

在开发基于 DSP 的嵌入式应用系统时,为了提高软件的可读写、可维护性,同时也为了加快开发进度。很多工程技术人员希望使用高级语言如 C 语言进行开发。但是由于定点 DSP 本身不能进行浮点运算,再加上 C 编译器的优化功能还不尽完善,用 C 语言编写的浮点运算程序在定点 DSP 上的执行效率远远低于人们的预期。为此许多工程技术人员在开发定点 DSP 程序时经常将浮点运算转换为定点运算,但是由程序员自己手工实现三角函数、对数等数学函数的定点运算是非常烦琐的工作,费时费力,而且程序的可靠性也难以保证。德州仪器公司推出的针对 TMS320C28x 和 TMS320C64x 系列芯片定点 DSP 的 IQmath 数学函数库,用定点算法优化实现了一些常用的数学函数,在一定程度上解决了这个问题。合理使用这些函数,可以大幅度提高 C 语言浮点运算程序的执行效率,明显缩短 DSP 的开发时间。

### 4.3.1 定点算法原理

浮点运算转换为定点运算的基础是浮点数的定点表示。浮点数定点表示的关键在于事先确定小数点在数据中的位置,也叫作数的定标。数的定标最常用的是 Q 表示法, Q 值表示假想的小数点右边的二进制数的位数, Q 值的确定与数的取值范围有关,并且影响到数值表示的精度。在不同的 Q 值下,同样位数的二进制数可以表示不同大小和不同精度的小数。例如在 Q15 下, 16 位二进制数的表示范围是  $-1 \sim +1$ , 而在 Q0 下则为  $-32\,768 \sim +32\,767$ 。当确定了一个浮点型变量的可能取值范围后,就可以选择最适合的 Q 值,使得该 Q 值下用定点数来进行运算时精度最高。在定点数的运算过程中,必须时刻注意运算的中间结果也不能超出当前 Q 值所确定的数的表示范围,如果有超出须及时进行调整,否则会得到不正确的运算结果。在 DSP 的 C 语言程序中使用了浮点数的定点表示后,就可以把浮点数转化为定点数进行运算,从而避开定点 DSP 进行浮点运算时效率不高的问题。

Q 格式: 小数点位于第  $n$  位元之右侧, 称为  $Q_n$  格式。

例如:

16 位元二进位无符号数: 0100 0010 1000 0001

在 Q0 格式下其表示的是:  $2^{14} + 2^9 + 2^7 + 2^0 = 17\,025(d)$

在 Q8 格式下其表示的是:  $2^6 + 2^1 + 2^{-1} + 2^{-8} = 66.503\,90 \sim (d)$

在 Q16 格式下其表示的是:  $2^{-2} + 2^{-7} + 2^{-9} + 2^{-16} = 0.259\,78 \sim (d)$

进行加法或减法时, Q 格式并不会影响运算法则, 两个 Q8 格式的小数相加, 所得到的数值仍是 Q8 格式。两个 Q6 格式相减, 所得到的数值仍是 Q6 格式。因此在定点数之加减运算并不因 Q 格式不同而有差异。不过可能会产生溢位 (overflow), 而且不同格式的数值不能直接相加减。

### 4.3.2 如何安装 IQmath 库

可以在 TI 网站上下载 TMS320C28x 和 TMS320C64x 系列芯片的 IQmath 库, 并进行安装, 安装后阅读 README.txt 可以了解到更多的详情。

64 系列 IQmath 安装完毕后，它将生成如图 4.2 所示的目录结构。

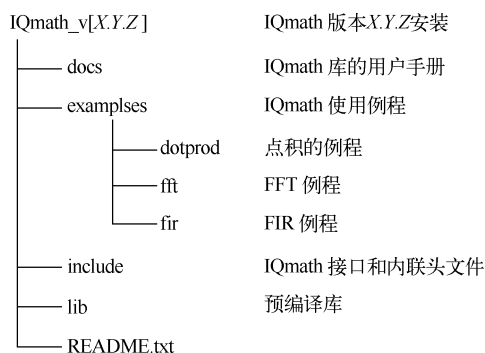


图 4.2 C64X IQmath 安装目录

### 4.3.3 如何使用 IQmath 库

TMS320C64x 系列输入/输出的 IQmath 函数通常是 32 位定点数，其固定点数的 Q 可以在 Q0 到 Q31 之间定义。开发者也可以使用自定义的 IQ 数据类型名，这更方便了开发者在应用程序中定义各种类型的 IQmath 数据类型。

表 4.19 总结了不同 Q 格式的 32 位定点数的表示范围和精度。所有 IQmath 都支持 Q1 ~ Q29 格式，此外 TMS320C64x 系列大多数函数也支持 Q0，Q1 ~ Q31 格式，具体可以参考表 4.20。

表 4.19 TMS320C64x 系列 Q 格式定义

<code>typedef int_iq; /* Fixed point data type;GLOBAL_Q format */</code>
<code>typedef int_iq31; /* Fixed point data type;Q31 format */</code>
<code>typedef int_iq30; /* Fixed point data type;Q30 format */</code>
<code>typedef int_iq29; /* Fixed point data type;Q29 format */</code>
<code>typedef int_iq28; /* Fixed point data type;Q28 format */</code>
<code>typedef int_iq27; /* Fixed point data type;Q27 format */</code>
<code>typedef int_iq26; /* Fixed point data type;Q26 format */</code>
<code>typedef int_iq25; /* Fixed point data type;Q25 format */</code>
<code>typedef int_iq24; /* Fixed point data type;Q24 format */</code>
<code>typedef int_iq23; /* Fixed point data type;Q23 format */</code>
<code>typedef int_iq22; /* Fixed point data type;Q22 format */</code>
<code>typedef int_iq21; /* Fixed point data type;Q21 format */</code>
<code>typedef int_iq20; /* Fixed point data type;Q20 format */</code>
<code>typedef int_iq19; /* Fixed point data type;Q19 format */</code>
<code>typedef int_iq18; /* Fixed point data type;Q18 format */</code>
<code>typedef int_iq17; /* Fixed point data type;Q17 format */</code>
<code>typedef int_iq16; /* Fixed point data type;Q16 format */</code>

续表

typedef int_iq15; /* Fixed point data type; Q15 format */
typedef int_iq14; /* Fixed point data type; Q14 format */
typedef int_iq13; /* Fixed point data type; Q13 format */
typedef int_iq12; /* Fixed point data type; Q12 format */
typedef int_iq11; /* Fixed point data type; Q11 format */
typedef int_iq10; /* Fixed point data type; Q10 format */
typedef int_iq9; /* Fixed point data type; Q9 format */
typedef int_iq8; /* Fixed point data type; Q8 format */
typedef int_iq7; /* Fixed point data type; Q7 format */
typedef int_iq6; /* Fixed point data type; Q6 format */
typedef int_iq5; /* Fixed point data type; Q5 format */
typedef int_iq4; /* Fixed point data type; Q4 format */
typedef int_iq3; /* Fixed point data type; Q3 format */
typedef int_iq2; /* Fixed point data type; Q2 format */
typedef int_iq1; /* Fixed point data type; Q1 format */
typedef int_iq0; /* Fixed point data type; Q0 format */

表 4.20 32 位定点数的表示范围和精度

Data Type	Range		Resolution/Precision
	Min	Max	
_iq31	-1	0.999 999 999	0.000 000 0005
_iq30	-2	1.999 999 999	0.000 000 001
_iq29	-4	3.999 999 998	0.000 000 002
_iq28	-8	7.999 999 996	0.000 000 004
_iq27	-16	15.999 999 993	0.000 000 007
_iq26	-32	31.999 999 985	0.000 000 015
_iq25	-64	63.999 999 970	0.000 000 030
_iq24	-128	127.999 999 940	0.000 000 060
_iq23	-256	255.999 999 981	0.000 000 119
_iq22	-512	511.999 999 762	0.000 000 238
_iq21	-1 024	1 023.999 999 523	0.000 000 477
_iq20	-2 048	2 047.999 999 046	0.000 000 954
_iq19	-4 096	4 095.999 998 093	0.000 001 907
_iq18	-8 192	8 191.999 996 185	0.000 003 815
_iq17	-16 384	16 383.999 992 371	0.000 007 629
_iq16	-32 768	32 767.999 984 741	0.000 015 259
_iq15	-65 536	65 535.999 969 482	0.000 030 518
_iq14	-131 072	131 071.999 938 965	0.000 061 035

续表

Data Type	Range		Resolution/Precision
	Min	Max	
_iq13	-262 144	262 143. 999 877 930	0. 000 122 070
_iq12	-524 288	524 287. 999 755 859	0. 000 244 141
_iq11	-1 048 576	1 048 575. 999 511 719	0. 000 488 281
_iq10	-2 097 152	2 097 151. 999 023 437	0. 000 976 563
_iq9	-4 194 304	4 194 303. 998 046 875	0. 001 953 125
_iq8	-8 388 608	8 388 607. 996 093 750	0. 003 906 250
_iq7	-16 777 216	16 777 215. 992 187 500	0. 007 812 500
_iq6	-33 554 432	33 554 431. 984 375 000	0. 015 625 000
_iq5	-67 108 864	67 108 863. 968 750 000	0. 031 250 000
_iq4	-134 217 728	134 217 727. 937 500 000	0. 062 500 000
_iq3	-268 435 456	268 435 455. 875 000 000	0. 125 000 000
_iq2	-536 870 912	536 870 911. 750 000 000	0. 250 000 000
_iq1	-1 073 741 824	1 073 741 823. 500 000 000	0. 500 000 000
_iq0	-2 147 483 648	2 147 483 648. 000 000 000	1. 000 000 000

#### 4.3.3.1 在 C 中调用一个 IQMath 函数

除了安装 IQmath 软件，在调用 IQmath 函数时还应：

- 在文件中包含 IQmath. h 文件；
- 将代码与 IQmath. h 连接；
- 在程序存储器中用正确的 CMD 文件放置 IQmath 代码段。

下面以 TMS320C64x 系列为例，其连接文件为：

```
MEMORY
{
    ERAM_01          :0 = 0X81000000,1 = 0X00100000
}
SECTIONS
{
    .data:IQmathTables    :   >  ERAM_01
    .text:IQmath          :   >  ERAM_01
}
```

下面代码包含在 IQmath. lib 中，调用 IQ25sin。

```
#include <IQmath . h>      /* Header file for IQmath routine */
#define PI3.14159F
_iq input,sin_out;        /* Definition of variables using IQmath datatype */
Void main(void)
```

```

{
    input = _IQ29(0.25 * PI); /* radians represented in Q29 format */
    sin_out = _IQ29sin(input);
}

```

#### 4.3.3.2 IQMath 函数命名规则

每一个 IQMath 函数都包括两种函数类型的 handles。

1. GLOBAL\_Q 函数，在这种格式中用到输入和输出。

例如：

```

➤ _IQsin(A)           /* High Precision SIN */
➤ _IQcos(A)           /* High Precision COS */
➤ _IQmpy(A,B)         /* IQ multiply with rounding */

```

2. Q 格式特殊函数，对于 TMS320C28x 系列主要是为了应用 Q1 ~ Q30 数据格式，对于 TMS320C64x 系列主要是为了应用 Q0 ~ Q31 数据格式。

例如：

```

➤ _IQ29sin(A)         /* High Precision SIN;input/output are in Q29 */
➤ _IQ25cos(A)         /* High Precision COS;input/output are in Q25 */
➤ _IQ20mpy(A,B)       /* fixed point multiply;input/output are in Q20 */

```

#### 4.3.3.3 选择 GLOBAL\_Q 格式

从一个应用到另一个应用，数字精度和动态范围的要求相差很大。IQMath 库为了方便在定点算法中程序的应用，不用事先确定数字精度。高精度导致低的动态量程，因此系统设计师在选用 GLOBAL\_Q 格式之前必须平衡二者之间的关系。

例如：

```

#ifndef GLOBAL_Q
#define GLOBAL_Q 24
#endif

```

#### 4.3.4 IQmath 库的函数功能

IQmath 库函数中包括以下几个部分。

- 格式转换函数：atoIQ、IQtoF、IQtoIQN 等。
- 算数函数：IQmpy、IQdiv 等。
- 三角函数：IQsin、IQcos、IQatan2 等。
- 数学函数：IQsqrt、IQisqrt 等。
- 其他：IQabs、IQsat 等。

对 64x 系列，在各函数的描述说明中采用了如下一些变量或符号。



- IQN 32 位定点 Q 格式数,  $N=0:31$ 。
- Int, INT, I32\_IQ 32bit 带符号数。
- \_iq 数据类型等同于 int, 代表了一个 32 位 GLOBAL\_Q。  
建议使用 \_iq 代替 int, 可以增加程序的可移植性。
- \_iqN 数据类型等同于 int, IQN 是一个 32 位数, 其中  $N=0:31$ 。
- A, B IQmath 函数或宏的输入操作数。
- F 浮点数输入, 比如:  $-1.232$ ,  $+22.433$ ,  $0.4343$ ,  $-0.32$ 。
- S 浮点字符串: “ $+1.32$ ”, “ $0.232$ ”, “ $-2.343$ ” 等。
- P 正数最大值。
- N 负数最大值。

#### 4.3.4.1 IQMath 格式转换函数

IQMath 格式转换函数主要用于格式之间的相互转换, 各个函数的具体功能见表 4.21。

表 4.21 TMS320C64x 系列格式转换函数

函 数	说 明	IQ 格式
_iq_FtoIQ(float F) _iqN_FtoIQN(float F)	浮点数转换为 IQN 数据值	Q = GLOBAL_Q Q = 1:31
_iq_IQ(int A)	将整数转换为 IQ 数据格式	Q = 0:31
Float_IQtoF(_iq A) Float_IQNtoF(_iqN A)	IQ 数据格式转换为浮点数	Q = GLOBAL_Q Q = 1:31
_iq_atoIQ(char *S) _iqN_atoIQN(char *S)	转换字符串为 IQ 数据格式	Q = GLOBAL_Q Q = 1:31
Int_IQint(_iq A) Int_IQNint(_iqN A)	提取 IQ 值的整数部分	Q = GLOBAL_Q Q = 0:29
_iq_IQfrac(_iq A) _iqN_IQNfrac(_iqN A)	提取 IQ 值的小数部分	Q = GLOBAL_Q Q = 1:31
_iqN_IQtoIQN(_iq A)	将 IQ 值转换为 IQN 值 (32bit)	Q = GLOBAL_Q
_iq_IQNtoIQ(_iqN A)	将 IQN 值 (32bit) 转换为 IQ 值	Q = GLOBAL_Q
_iqY_IQXtoIQY(_iqX A, int x, int y)	将 IQX 的值转换为 IQY 的值	Q = 1:31

下面对 TMS320C64x 系列 IQMath 格式转换函数的使用举例说明。

例如: 把浮点数常量或变了转换成为 IQ 数据类型

```
Float x = 3.343;
_iq y1;
_iq23 y2;
IQMath(类型1): y1 = _FtoIQ(x)
IQMath(类型2): y2 = _FtoIQ23(x)
```

例如: 用 IQMath 方法实现方程

浮点数方程为:  $Y = M \times 1.26 + 2.345$

其 IQMath 对应的方程 (类型 1):  $Y = \_IQmpy(M, \_FtoIQ(1.26)) + \_FtoIQ(2.345)$

其 IQMath 对应的方程 (类型 2):  $Y = \_IQ23mpy(M, \_FtoIQ23(1.26)) + \_FtoIQ23$

## (2.345)

例如：分离出两个 IQ 数值的整数部分和小数部分

```
_iq Y0 = 2.3456;
_iq Y1 = -2.3456;
Long Y0int, Y1int;
_iq Y0frac, Y1frac;
Y0int = Iqint(Y0);           //Y0int = 2
Y1int = Iqint(Y1);           //Y1int = -2
Y0frac = _Iqfrac(Y0);        //Y0frac = 0.3456
Y1frac = _Iqfrac(Y1);        //Y1frac = -0.3456
```

#### 4.3.4.2 IQMath 算数函数

IQMath 算数函数主要包含一些常用的算数运算，各个函数的具体功能见表 4.22。

表 4.22 TMS320C64x 系列算数函数

函 数	说 明	IQ 格式
_iq _IQmpy( _iq A, _iq B) _iqN _IQNmpy( _iqN A, _iqN B)	IQ 乘法	Q = GLOBAL_Q Q = 1:31
_iq _IQrmpy( _iq A, _iq B) _iqN _IQNrmpy( _iqN A, _iqN B)	四舍五入的 IQ 乘法	Q = GLOBAL_Q Q = 1:31
_iq _IQrsmpy( _iq A, _iq B) _iqN _IQNrsmpy( _iqN A, _iqN B)	带四舍五入和饱和处理的 IQ 乘法	Q = GLOBAL_Q Q = 1:31
_iq _IQmpyI32( _iq A, int B) _iqN _IQNmpyI32( _iqN A, int B)	IQ 格式与整形相乘	Q = GLOBAL_Q Q = 1:30
int _IQmpyI32int( _iq A, int B) int _IQNmpyI32int( _iqN A, int B)	IQ 格式与整形相乘，返回整数部分	Q = GLOBAL_Q Q = 0:30
int _IQmpyI32frac( _iq A, int B) int _IQNmpyI32frac( _iqN A, int B)	IQ 格式与整形相乘，返回小数部分	Q = GLOBAL_Q Q = 0:30
_iq _IQmpyIQX( _iqN1 A, N1, _iqN2 B, N2) _iqN _IQNmpyIQX( _iqN1 A, N1, _iqN2 B, N2)	IQ 格式不同的两个数相乘	Q = GLOBAL_Q Q = 0:31
_iq _IQdiv( _iq A, _iq B) _iqN _IQNdiv( _iqN A, _iqN B)	定点数除法	Q = GLOBAL_Q Q = 0:31

下面对 TMS320C64x 系列 IQMath 算数函数的使用举例说明。

IQ 乘法是两个 IQ 值相乘，它并不表现出饱和以及四舍五入的格式。在大多数情况下，两个 IQ 值相乘不会超出 IQ 变量的范围。由于该操作使得运行周期最小并且代码最少，所以经常使用。当为全局 IQ 格式（即 IQ 格式等于 GLOBAL\_Q 时），使用 \_iq\_IQmpy( \_iq A, \_iq B) 函数，如果为特定 IQ 格式，使用 \_iqN\_IQNmpy( \_iq NA, \_iq NB) 函数。

例如：在 GLOBAL\_Q 格式下进行  $Y = M * X + B$  的计算，不带饱和和四舍五入。

```
_iq Y, M, X, B;
Y = _Iqmpy(M, X) + B;
```

例如：在 Q10 格式下进行  $Y = M \times X + B$  的计算，不带饱和和四舍五入，假设 M、X 和 B 是 IQ10 格式。

```
_iq Y,M,X,B;
Y = _Iq10mpy(M,X) + B;
```

带四舍五入以及饱和处理的 IQ 乘法, 是对两个 IQ 数据进行乘法运算, 并进行四舍五入以及饱和处理。当计算有可能超出 IQ 变量范围时, 在存储结果之前对结果四舍五入以及进行饱和处理, 达到 IQ 变量范围的最大值。

例如: 在 Q26 格式下进行  $Y = M \times X$  的计算, 带饱和以及四舍五入处理。

```
_iq26 Y,M,X;
M = _FtoIQ(-10.9); //M = -10.9
X = _FtoIQ(4.5); //X = 4.5
Y = _IQrsmpr(M,X); //Y = ~32,输出已饱和,故输出最大值
```

例如: 在 GLOBAL\_Q 格式下进行  $Y = M \times X$  的计算, 带饱和以及四舍五入处理 (假定 GLOBAL\_Q = 26)。

```
_iq Y,M,X;
M = _IQ26(10.9); //M = 10.9
X = _IQ26(4.5); //X = 4.5
Y = _IQ26rsmpr(M,X); //Y = -32,输出已饱和,故输出最小值
```

除了将相同格式的数进行乘法外, 还可以使用 IQNmpyIQX 函数对两个不同 Q 格式的数进行相乘。

例如: 计算等式  $Y = X0 \times C0 + X1 \times C1 + X2 \times C2$ , 其中  $X0$ 、 $X1$  和  $X2$  是 IQ30 (范围为  $-2 \sim +2$ ),  $C0$ 、 $C1$  和  $C2$  是 IQ28 (范围为  $-8 \sim +8$ )。Y 的最大范围将是  $-48 \sim +48$ , 因此应该用小于 IQ25 的格式来存储结果。

情况 1: GLOBAL\_Q = IQ25

```
_iq30 X0,X1,X2; //所有变量为 IQ30 格式
_iq28 C0,C1,C2; //所有变量为 IQ28 格式
_iq Y; //结果 GLOBAL_Q = IQ25 格式
Y = _IQMPYIQX(X0,30,C0,28);
Y += _IQMPYIQX(X1,30,C1,28);
Y += _IQMPYIQX(X2,30,C2,28);
```

情况 2: 特定 IQ 的计算

```
_iq30 X0,X1,X2; //所有变量为 IQ30 格式
_iq28 C0,C1,C2; //所有变量为 IQ28 格式
_iq25 Y; //结果 IQ25 格式
Y = _IQ25MPYIQX(X0,30,C0,28);
Y += _IQ25MPYIQX(X1,30,C1,28);
Y += _IQ25MPYIQX(X2,30,C2,28);
```

使用牛顿 - 拉夫逊方法 (Newton - Raphson) 对两个 IQN 的数进行相除, 并给出 32bit 的商 (IQN 格式)。

例如: 计算  $1/1.5 = 0.666$ , 假设在 Iqmath 头文件里, GLOBAL\_Q 被设为 Q28 格式。

```
#include <IQmath.h> /* IQmath 函数的头文件 */

_iq in1 out1;
_iq28 in2, out2;
Void main (void)
{
    in1 = FtoIQ (1.5);
    out1 = _IQdiv (_FtoIQ (1.0), in1);
    in2 = _IQ28 (1.5);
    out2 = _IQ28div (_FtoIQ28 (1.0), in2);
}
```

#### 4.3.4.3 IQMath 三角函数

IQMath 三角函数主要包含一些常用的三角运算, 各个函数的具体功能见表 4.23。

表 4.23 TMS320C64x 系列三角函数

函 数	说 明	IQ 格式
_iq _IQsin( _iq A) _iqN _IQNsin( _iqN A)	高精度正弦函数 (输入单位: 弧度)	Q = GLOBAL_Q Q = 1: 29
_iq _IQsinPU( _iq A) _iqN _IQNsinPU( _iqN A)	高精度正弦函数 (输入单位: 标准值 <sup>(1)</sup> )	Q = GLOBAL_Q Q = 1: 30
_iq _IQasin( _iq A) _iqN _IQNasin( _iqN A)	高精度反正弦函数 (输出单位: 弧度)	Q = GLOBAL_Q Q = 1: 29
_iq _IQcos( _iq A) _iqN _IQNcos( _iqN A)	高精度余弦函数 (输入单位: 弧度)	Q = GLOBAL_Q Q = 1: 29
_iq _IQcosPU( _iq A) _iqN _IQNcosPU( _iqN A)	高精度余弦函数 (输入单位: 标准值)	Q = GLOBAL_Q Q = 1: 30
_iq _IQacos( _iq A) _iqN _IQNacos( _iq A)	高精度反余弦函数 (输出单位: 弧度)	Q = GLOBAL_Q Q = 1: 29
_iq _IQatan2( _iq A, _iq B) _iqN _IQNatan2( _iqN A, _iqN B)	第四象限反正切函数, 输出为 (A/B) 的值 (输出单位: 弧度)	Q = GLOBAL_Q Q = 1: 29
_iq _IQatan2PU( _iq A, _iq B) _iqN _IQNatan2PU( _iqN A, _iqN B)	第四象限反正切函数, 输出为 (A/B) 的值 (输出单位: 标准值)	Q = GLOBAL_Q Q = 1: 30
_iq _IQatan( _iq A) _iqN _IQNatan( _iqN A)	第四象限反正切函数, (输出单位: 弧度)	Q = GLOBAL_Q Q = 1: 29

注 (1): 一个标准值为  $(2p / 512)$  弧度

下面对 TMS320C64x 系列 IQMath 三角函数的使用举例说明。

定点正弦函数使用查询表项目中的查询表和泰勒级数展开来计算输入的正弦值。

例如: 假设 GLOBAL\_Q 在 IQmath 头文件中设置为 Q29 格式, 计算  $\sin(0.25 \times \pi) = 0.707$ 。

```
#include <IQmath.h>
#define PI 3.14156
```

```

_iq in1,out1;
_iq28 in2,out2;
Void main(void)
{
    in1 = _FtoIQ(0.25 * PT);    /* in1 = 0.25 × π × 229 = 1921FB54h */
    out1 = _IQsin(in1);        /* out1 = sin(0.25 × π) × 229 = 16A09E66h */
    in2 = _FtoIQ29(0.25 * PT); /* in2 = 0.25 × π × 229 = 1921FB54h */
    out2 = _IQ29sin(in2);      /* out2 = sin(0.25 × π) × 229 = 16A09E66h */
}

```

第四象限定点反正切函数 IQNatan2，输出为弧度值，大小为  $-\pi \sim +\pi$ 。

例如：假设 GLOBAL\_Q 在 IQmath 头文件中设置为 Q29 格式，求  $\tan^{-1}[\sin(\pi/5), \cos(\pi/5)] = \pi/5$ 。

```

#include <IQmath.h>
#define PI 3.14156
_iq xin1,yin1,out1;
_iq29 xin2,yin2,out2;
Void main(void)
{
    xin1 = _FtoIQ(0.809);      /* xin1 = cos(π/5) × 229 = 19E3779Bh */
    yin1 = _FtoIQ(0.5877);     /* yin1 = cos(π/5) × 229 = 12CF2304h */
    out1 = _IQatan2(yin1,xin1); /* out1 = π/5 × 229 = 141B2F76h */
    xin2 = _FtoIQ(0.809);      /* xin2 = cos(π/5) × 229 = 19E3779Bh */
    yin2 = _FtoIQ(0.5877);     /* yin2 = cos(π/5) × 229 = 12CF2304h */
    out2 = _IQ29atan2(yin2,xin2); /* out2 = π/5 × 229 = 141B2F76h */
}

```

#### 4.3.4.4 IQMath 数学函数

IQMath 数学函数主要包含一些常用的数学运算，各个函数的具体功能见表 4.24。

表 4.24 TMS320C64x 系列数学函数

函 数	说 明	IQ 格式
_iq _IQsqrt( _iq A) _iqN _IQNsqrt( _iqN A)	高精度平方根函数	Q = GLOBAL_Q Q = 0:30
_iq _IQisqrt( _iq A) _iqN _IQNisqrt( _iqN A)	高精度平方根的倒数函数	Q = GLOBAL_Q Q = 0:30
_iq _IQmag( _iq A, _iq B) _iqN _IQNmag( _iqN A, _iqN B)	求模运算: $\sqrt{A^2 + B^2}$	Q = GLOBAL_Q Q = 0:30
_iq _IQexp( _iq A) _iqN _IQNexp( _iqN A)	指数运算函数	Q = GLOBAL_Q Q = 1:30
_iq _IQlog( _iq A) _iqN _IQNlog( _iqN A)	对数运算函数	Q = GLOBAL_Q Q = 1:30
_iq _IQpow( _iq A) _iqN _IQNpow( _iqN A) Q = 1	幂运算函数: $\text{pow} = A^B$	Q = GLOBAL_Q Q = 1:30

下面对 TMS320C64x 系列 IQMath 数学函数的使用举例说明。

定点平方根函数 IQNsqrt，用来计算平方根，其使用的是查表和牛顿 - 拉夫逊逼近算法。

例如：假设 GLOBAL\_Q 在 IQmath 头文件中设置为 Q29 格式，计算  $\sqrt{1.8} = 1.34164$ 。

```
#include <IQmath.h>
_iq in1,out1;
_iq30 in2,out2;
Void main(void)
{
    in1 = _FtoIQ(1.8);           /* in1 = 1.8 × 230 = 73333333h */
    out1 = _IQsqrt(x);           /* out1 =  $\sqrt{1.8} \times 2^{30} = 55DD7151h$  */
    in2 = _FtoIQ30(1.8);        /* in2 = 1.8 × 230 = 73333333h */
    out2 = _IQ30sqrt(x);        /* out2 =  $\sqrt{1.8} \times 2^{30} = 55DD7151h$  */
}
```

求模运算函数 IQNmag 用来计算两个正交向量的幅值， $\text{Mag} = \sqrt{A^2 + B^2}$ 。计算达到了更高的精确度，同时避免了使用 \_IQsqrt 函数式遇到的溢出问题。

例如：求复数的幅值（假设 GLOBAL\_Q = 28）。

```
#include <IQmath.h>      /* IQmath 函数头文件 */
_iq real1,imag1,mag1;    /* 复数 = real1 + j * imag1 */
_iq28 real2,imag2,mag2;  /* 复数 = real2 + j * imag2 */
Void main(void)
{
    real1 = _FtoIQ(4.0);
    imag1 = _FtoIQ(4.0);
    mag1 = _IQmag(real1,imag1); /* 在 IQ28 格式下 mag1 = 5.6568 */
    real2 = _FtoIQ28(7.0);
    imag2 = _FtoIQ28(7.0);
    mag2 = _IQ28mag(real2,imag2); /* 在 IQ28 格式下,饱和至最大值 mag2 = ~8.0 */
}
```

#### 4.3.4.5 IQMath 其他函数

IQMath 除了以上介绍的四种函数外，还有限幅函数和绝对值函数。各个函数的具体功能见表 4.25。

表 4.25 TMS320C64x 系列限幅函数和绝对值函数

函 数	说 明	IQ 格式
I32 _IQsat( I32 A) I32 _IQNsat( I32 A)	IQ 数值的限幅函数	int32
_iq _IQabs( _iq A) _iqN _IQNabs( _iqN A)	IQ 数值的绝对值	Q = GLOBAL_Q Q = 0: 31



限幅函数能够将一个 IQ 值限幅在给定的正负极限，经常应用在计算溢出的场合。

例如：以饱和方式计算线性方程  $Y = M \times X + B$ 。

所有变量均为 GLOBAL\_Q = 26，然而变化范围可能产生溢出。为此，应采取中间操作使 IQ = 20，并在将结果转化回适当的全局 Q 格式之前将其饱和化。

```
iq Y,M,X,B;    //GLOBAL_Q = 26( + / - 32 range)
_iq20 temp;     //IQ = 20( + / - 2048 range)
temp = _IQ20mpy( _IQtoIQ20( M ), _IQtoIQ20( X ) ) + _IQtoIQ20( B );
temp = _IQsat( temp, _IQtoIQ20( NAX_IQ_POS ), _IQtoIQ20( NAX_IQ_NEG ) );
Y = _IQ20toIQ( temp );
```

绝对值函数 IQNabs 用于计算一个 IQ 数值的绝对值。

例如：计算 3 个 IQ 数的绝对值之和（假设 GLOBAL\_Q = 28）。

```
_iq xin1,xin2,xin3,xsum;
_iq20 yin1,yin2,yin3,ysum;
xsum = _IQabs( X0 ) + _IQabs( X1 ) + _IQabs( X2 );
xsum = _IQ28abs( X0 ) + _IQ28abs( X1 ) + _IQ28abs( X2 );
```

## 4.4 DSP 的图像处理库 IMGLIB

TI 公司提供的 IMGLIB 库文件包括了很多图像和视频处理函数，所有函数都是对 C 语言编程进行了优化。该库包括一些可以使用 C 语言调用，且已经过汇编优化的图像和视频处理子程序。在对图像处理时间十分敏感、计算量很大的实时系统中，可以使用这些已经经过计算优化的函数来提高执行速度。用户可以根据产品的特点，修改库的源程序满足自己的要求。这些源程序可以在 Code Composer Studio 软件的安装目录下找到。

目前有 TMS320C55x imglib 和 TMS320C64x imglib，分别适用于 TI C5000 和 C6000 系列的 DSP 设备。

本节均以 TMS320C64x IMGLIB 为例，对 IMGLIB 安装、使用等进行说明。

### 4.4.1 如何安装和调用 IMGLIB 库

IMGLIB 提供一个自安装可执行文件，imglibc64plus-2. -xx -setup. exe，安装 IMGLIB 默认位置是 C:\CCStudio\_v3.3\c64plus，用户可以修改安装目录的位置。安装完毕后，它将生成如图 4.3 所示的目录结构。

其中 host 文件夹是供 VC 使用，target 文件夹是供 CCS 使用。

调用 IMGLIB 库步骤如下。

第一步：在项目中添加 IMGLIB 库文件。

VC 中配置如下：选中菜单栏的“工具 - 选项 - 项目和解决方案 - VC ++ 目录”，在“显示以下内容的目录”一栏选中“库文件”，在里面添加 imglib2\_host.lib 所在的路径“C:\CCStudio\_v3.3\c64plus - imglib\_2\_02\_00\_00\lib\host”，然后选中菜单栏的“项目 - xx 项目

IMGLIB2	
+--build	project files to builds host/target lib
--host	
--target	
+--docs	library documentation
+--include	Required include files
+--kernels	Kernel sources
--asm	
--c	
--intrinsic	
--serial_asm	
+--lib	host and target library
--host	
--target	
+--test_drivers	test bench with reference input/output vectors
--drivers	
+--set of test-cases	
+--README.txt	Top-level README file
+--TI_license.pdf	License Agreement file

图 4.3 C64X IMGLIB 安装目录

属性 - 连接器 - 输入”，在“附加依赖库”一栏加入 imglib2\_host.lib。

CCS 中配置如下：右击项目，选择“Build Options - Linker - Libraries”，在 Search Path 一栏添加其路径“C:\CCStudio\_v3.3\c64plus - imglib\_2\_02\_00\_00\lib\target”，在 incl. Libraries 一栏添加库文件名 limglib2.lib。

第二步：调用相应函数时，添加相应的头文件，例如：要调用函数 IMG\_dilate\_bin\_cn()，需源文件在 main.c 中添加#include "IMG\_dilate\_bin.h"

#### 4.4.2 IMGLIB 库的函数功能

IMGLIB 里的软件程序主要分为三类：压缩与解压缩、图像分析、图像滤波/格式转换。

##### 4.4.2.1 图像压缩、解压缩子程序

该部分包括了标准图像压缩/解压缩算法子程序，如 JPEG、MPEG Video 和 H.26x 等算法。

函数：

- IMG\_fdct\_8x8;
- IMG\_idct\_8x8。

前向和反转离散余弦变换函数。在大多数标准压缩算法中都使用离散余弦变换函数，如 JPEG 编码/解码、MPEG 视频编码/解码和 H.26X 编码/解码。

函数：

- IMG\_mad\_8x8;

- IMG\_mad\_16 × 16;
- IMG\_sad\_8 × 8;
- IMG\_sad\_16 × 16。

最小绝对差函数和绝对差求和函数。利用这些函数可以提高运动图像识别算法性能，在视频编码系统中，运动图像识别算法是得到最大计算加强优化。采用 TI 提供的函数可以使系统中算法性能得到显著改善。

函数：

- IMG\_mpeg2\_vld\_inter;
- IMG\_mpeg2\_vld\_intra。

MPEG-2 可变长度解码函数。该函数提供了一个高集成度和高效率解决方案，该方案优化了 MPEG-2 代码 intra 和 non-intra 宏块的可变长度解码、run-length expansion、反转扫描、dequantization、saturation 和 mismatch 控制。任何 MPEG-2 视频解码系统的性能依赖于每个解码步骤的高效实现。

函数：

- IMG\_quantize。

量化函数。量化是许多图像视频压缩系统中的积分步骤，包括 DCT 压缩算法基础之上各种变异算法，例如 JPEG、MPEG 和 H.26X 等算法。该子程序可以提高量化步骤的速度和性能。

函数：

- IMG\_wave\_horz;
- IMG\_wave\_vert。

小波变换函数。在 JPEG2000 和 MPEG-4 等算法中，小波处理得到了广泛的应用，并将发展成为一种标准，典型应用于提高静止图像压缩的性能方面，而且在各种图像压缩系统中都是建立在小波处理基础之上。IMG\_wave\_horz 和 IMG\_wave\_vert 函数用于计算水平和垂直小波变换。利用这两个函数可以计算图像数据 2 维小波变换。该子程序在文档约束之内使用非常灵活，可以满足大范围的特殊小波变换和图像维数。

#### 4.4.2.2 图像处理子程序

这部分包括了应用于图像处理的常用函数，下面对函数进行简要说明。

函数：

- IMG\_boundary;
- IMG\_perimeter。

边界和周界函数。它们通常在结构视觉应用中作为结构算子。

函数：

- IMG\_dilate\_bin;
- IMG\_erode\_bin。

膨胀和腐蚀函数。这两个函数是图像学算子，通常用于提高二进制图像扩大和二进制图像侵蚀算法效果。扩大和侵蚀在图像处理操作中具有基础的意义，比如打开和关闭都可以从扩大和侵蚀中建立起来。这些函数在机器视觉和医学成像方面非常有用。

函数：

➤ IMG\_histogram。

直方图计算函数。直方图用来生成图像的柱状图，图像的直方图是一个图像亮度级的统计。例如，对于一个 8 位像素亮度级别的灰度图像，直方图将包括对应可能的 256 个像素亮度的 256bins。每一个 bin 包含图像中像素点的个数，尤其是亮度值。

函数：

➤ IMG\_sobel。

Sobel 边界检测函数。在机器视觉系统中通常使用边界检测技术。在许多算法中都存在边界检测技术，最通用的是 Sobel 边界检测。IMG\_sobel 子程序提供了一个边界检测算法优化执行的子程序。

函数：

➤ IMG\_thr\_gt2max；

➤ IMG\_thr\_gt2thr；

➤ IMG\_thr\_le2min；

➤ IMG\_thr\_le2thr。

阈值函数。在图像和视频处理系统中图像阈值操作的不同形式满足不同的图像处理需求。例如，一种阈值可以用于把灰度图像数据转化为二进制图像数据，以用于二进制形态处理。另一个阈值可以用于剪裁图像数据以便得到期望的范围。在机器视觉应用中，阈值用于简单的分割。

#### 4.4.2.3 图像滤波处理算法

这部分包括了用于图像滤波和格式转化操作的几个函数，下面对函数进行简要说明。

函数：

➤ IMG\_conv\_3 × 3。

卷积函数。该函数用于普通图像 3 × 3 滤波，比如图像平滑和锐化处理等方法。

函数：

➤ IMG\_corr\_3 × 3；

➤ IMG\_corr\_gen。

相关性函数。该函数用于图像匹配操作。在机器视觉、医学成像和安全/保卫方面，图像匹配处理是非常有用的。库中提供了两个相关性函数，IMG\_corr\_3 × 3 函数实现对 3 × 3 像素区域高度优化相关性处理。IMG\_corr\_gen 是一个更普通的版本，能够对用户特殊的像素区域大小进行相关性处理。

函数：

➤ IMG\_errdif\_bin。

二进制值输出误差扩散函数。最常用的误差扩散算法是 Floyd - Steinberg 算法，广泛用于印花行业中，此函数中对该算法进行了优化。

函数：

➤ IMG\_median\_3 × 3。

中值滤波函数。在图像恢复处理中，中值滤波用于使成像中的脉冲噪声产生的影响降到最低，该处理方法可以保护脉冲影响十分严重的局部区域，此函数对中值滤波提供了优化实现。

函数:

➤ IMG\_pix\_expand;

➤ IMG\_pix\_sat。

像素扩展和截断函数。IMG\_pix\_expand 子程序用于通过零扩展技术把 8 位像素点扩展到 16 位; IMG\_pix\_sat 用于把 16 位有符号数转换成 8 位无符号数。通常用于处理其他子程序的输入和输出子程序, 例如水平和垂直缩放子程序。

函数:

➤ IMG\_ycbcr422p\_rgb565。

色彩空间转换函数。该子程序实现图像从 ycbcr 格式到 RGB 格式的转化, 以实现在 MPEG 和 JPEG 解码系统中的视频数据在 RGB 显示器中的播放。

函数:

➤ IMG\_yc\_demux\_bel6;

➤ IMG\_yc\_demux\_lel6。

ycbcr 色彩空间分离函数。此函数开辟一个 little endian 或 big endian 格式的 YcrYCb 彩色图像缓冲区, 将 ycbcr 图像数据分离为 y、cb、cr 三通道数据。

### 4.4.3 IMGLIB 函数使用举例

以二值图像的  $3 \times 3$  腐蚀函数 IMG\_dilate\_bin\_cn() 为例, 用 4.4.1 节介绍的方法添加库文件和头文件后, 可以在 C:\CCStudio\_v3.3\c64plus - imglib\_2\_02\_00\_00\kernels\c 文件夹内查看其源代码 IMG\_dilate\_bin\_c.c, 了解其使用方法。

函数定义为:

```
void IMG_dilate_bin_cn
(
    const unsigned char * in_data,      //输入数据
    unsigned char      * out_data,      //膨胀后的输出数据
    const char         * mask,          //结构模板
    int cols            //图像的列数(little endian 模式)
)
```

假设原二值图像高为 HEIGHT, 宽为 WIDTH, 调用膨胀函数前, 需将二值图像数据转换为 little endian 模式, 转换后一个字节即可表示 8 个像素, 图像的宽变为 WIDTH/8。

main 函数中定义了以下数组。

```
unsigned char imgbinary[ WIDTH * HEIGHT]      //二值图像数据(普通模式)
unsigned char imglit[ WIDTH * HEIGHT/8]       //二值图像数据(little endian 模式)
unsigned char imglitdilate[ WIDTH * HEIGHT/8] //膨胀后的数据(little endian 模式)
unsigned char imgdilate[ WIDTH * HEIGHT/8]    //膨胀后的数据(普通模式)
```

以下是模式转换代码, 函数 NormToLittle 将数据由普通模式转为 little endian 模式, 函数 LittleToNorm 将数据由 little endian 模式转为普通模式。

```

void NormToLittle(unsigned char * datain,unsigned char * dataout)
//LITTLE ENDIAN 编号,编号大的在左边高位,编号小的在右边低位
{
    int i,k,m;
    for(i=0;i<WIDTH * HEIGHT;i++)
    {
        k=i/8;
        m=i%8;
        dataout[k] |= datain[i] << m;
    }
}

void LittleToNorm(unsigned char * datain,unsigned char * dataout)
{
    int i,k,m;
    for(i=0;i<WIDTH * HEIGHT;i++)
    {
        k=i/8;
        m=i%8;
        dataout[i] = (datain[k] >> m) & 0x01;
    }
}

main 函数中调用膨胀函数的代码为:
const char mask[9] = {1,1,1,1,1,1,1,1,1};
unsigned char * pt1 = imglit;
unsigned char * pt2 = imglitdilate + WIDTH/8;

NormToLittle( imgbinary, imglit );
LittleToNorm( imglitdilate, imgdilate );
for( int i=0;i<HEIGHT-2;i++)
{
    IMG_dilate (pt1, pt2, mask, WIDTH/8);
    pt1 += WIDTH/8;
    pt2 += WIDTH/8;
}

```

## 4.5 DSP 的音频、视频和语音编解码器

在开发基于 DSP 的应用系统时,开发者可能需要用到音频、视频和语音编解码器。TI 公司提供了免费的音频、视频和语音编解码器。开发者可以直接使用 TI 提供的编解码器,从而节约了开发时间。图 4.4 列出了针对不同的 TI 处理器提供的各种优化后编解码器。这些版本可能会或不会被集成到每个平台最新的软件开发工具包中。开发者如果要构建和测试最新的软件开发工具包,可以到 TI 网站上下载。



Codecs	Target Hardware											
	C64x+™ DSP core-based devices*	Optimized for DM814x	Optimized for DM8168	Optimized for DM3730	Optimized for DM36x (DM365, DM368)**	Optimized for DM355S**	Optimized for OMAP35x	Optimized for DM646x	Optimized for DM644x	Optimized for DM643x	Optimized for DM648	C55x™
Video & Imaging												
H.264 Video Decoder	●	●	●	●	●		●	●	●		●	
H.264 Video Encoder	●	●	●	●	●		●	●	●	Request PROD	●	
VICP								● (HdVICP)				
VC1 Encoder		●	●									
VC1 Decoder		●	●		●							
JPEG Imaging Decoder	●			●	●	In DVSDK					●	
JPEG Imaging Encoder	●			●	●	In DVSDK					●	
MPEG-2 Video Decoder	●	●	●	● (MP)	●			●				
MPEG-2 Video Encoder		●	●		●				Request EVAL			
MPEG-4 Video Decoder	●	●	●	● (SP)	●	In DVSDK		●	Request EVAL (ASP)		●	
MPEG-4 Video Encoder	●	●	●	● (SP)	●	In DVSDK	●	●	● — Request EVAL (ASP)			
Audio												
AAC Audio Decoder	●	●	●	●	●	Request PROD						●
AAC Audio Encoder	●				●	Request PROD						●
AEC					●	Request PROD						
MPEG-2 L1/L2 Audio Decoder									Request EVAL			
MPEG-2 L1/L2 Audio Encoder									Request EVAL			
MP3 Audio Decoder	●	●	●	●	●	Request PROD — Also on DVSDK page						●
MP3 Audio Encoder					●	Request PROD			Request EVAL (HS/LIS)			
WMA Audio Decoder/Encoder	●				●	Request PROD						● (Dec)
Speech/Voice												
G.711 Decoder/Encoder	●	● (Dec)	● (Dec)	●	●	Open Source — Also in DVSDK						
G.722 Decoder/Encoder	●											●
G.726 Decoder/Encoder	●											●

图 4.4 TI 各个 DSP 系列编、解码器

下面将对常用的音频、视频和语音编解码器进行简要的介绍。

### 4.5.1 视频编解码器

TI 的视频编解码器主要包括了：MPEG - 4 视频编解码器、MPEG - 2 视频编解码器、VC1 编解码器和 H. 264 视频编解码器。

#### 4.5.1.1 MPEG - 4/H. 263 编解码器

TI 的 MPEG - 4/ H. 263 编码器和解码器 (codec) 软件, 符合国际标准化组织 (ISO) 和国际电联 (IEC) 所发布的 MPEG - 4 标准, 也符合国际电信联盟 (ITU) 的 H. 263 标准。

MPEG - 4 最初是由移动图像专家组 (MPEG, Moving Picture Experts Group) 开发的。继 MPEG - 1 和 MPEG - 2 之后, MPEG - 4 是一种非常先进的、用于多媒体表达和发送的标准。

MPEG - 4 提供多种压缩选择, 包括由无线设备发送使用的低带宽格式, 以及演播室处理使用的高带宽格式。MPEG - 4 SP (Simple Profile) 用于手机或其他类似的处理能力有限的设备。

ITU 开发 H. 263 标准的原意是用于视频会议发送的压缩, 由于其优异的性能, 已经广泛地应用于视频通信的编码标准, 也成为多种网络传输标准的一部分, 包括 ITU - T H. 324 (PTSN)、H. 320 (ISDN)、H. 310 (BISDN)、H. 323 以及 3GPP。

主要特性如下。

编码器:

- 符合 MPEG - 4 Simple Profile (SP) Levels 0/1/2/3/4A/5A;
- 支持 H. 263 baseline profile (Profile 0);
- 使用 eXpressDSP 软件;
- 支持标准的 TMN5、TM5 以及 VW4 速率控制算法, 以及自有标准的速率控制;
- 支持任意分辨率, 最高可达 HD (1 280 × 720), 包括标准的图像大小, 诸如 SQCIF、QCIF、CIF、QVGA、VGA;
- 支持半像素内插, 以便运动估计;
- 支持 I 帧 QP 参数的设置, 指定 P 帧的最大和最小 QP 值;
- 支持 I 帧的插入, 运行时改变视频包的大小;
- 编码器产生的比特流, 符合 MPEG - 4 标准的视频缓存验证;
- 帧的输入格式, 可以是 422i 或 420;
- 可以用 DaVinci 仿真器 CCS IDE 来验证。

解码器:

- 支持 MPEG - 4 SP Level 5;
- 支持 H. 263 Profile 0;
- 符合 eXpressDSP 软件标准;
- 输出格式 YUV420 / YUV422;
- 所包括的优化 I 和 P 帧流, 在每秒 30 帧 (fps) 时支持 720p;
- 解码器可以做彩色格式转换, 输出 YUV 格式的 420 和 422 隔行扫描;
- 解码器的库, 符合 eXpressDSP 软件标准, 并符合除了 DMA 指导方针的大部分 xDIAS 算法指导方针。

#### 4.5.1.2 H.264 Baseline Profile (BP) 编码器和解码器

H.264 是国际电信联盟 (ITU, International Telecommunications Union) 发布的一个视频压缩标准,其目的是,在比以往低的比特率下,提供高质量视频的压缩技术。该标准具有高度的灵活性,可以实现各种比特率,以及各种不同的分辨率。因此,H.264 在各种网络和系统中工作良好,包括移动媒体播放器、手机等。

主要特性如下。

- H.264 Baseline Profile (BP) 可以到 Level 3。
- 双用户可配置的速率控制方案:DCES\_TM5 和 PLR。
- 任意分辨率,可达 SD (720 × 480),包括标准的图像大小,诸如 SQCOF、QCIF、CIF、QVGA、VGA。
- 用户可以定义的运动估计的设置,以便控制编码速度和质量之间的平衡。
- 运动估计的 1/4 像素内插。
- 用户可以配置的 GOP 长度。
- 通过自由的运动搜索,预测丢失数据,来改善网络的性能。
- 解锁 (两种模式: on、off)。
- 可以在输入格式 YUV 平面和 YUV 422 隔行扫描之间做选择。
- eXpressDSP 软件兼容的解码器。

#### 4.5.1.3 MPEG-2 解码器

MPEG-2 是一种由 ISO 定义的标准,用于数字视频的编解码,支持各种分辨率的视频编码,包括众所周知的高清晰度电视 (HDTV)。

主要特性如下。

- 支持 ISO/IEC 13818-2 标准的 MP@ML 性能。
- 输出 YUV 420/422 格式。
- 支持隔行扫描和逐行扫描格式。
- 符合 ISO/IEC 13818-4, based 标准。
- 支持隔行和逐行解码。
- 支持 720p/1080i 解码。
- 解码器的 API 支持基于格式的解码。
- 只支持分量流输入格式。
- 支持综合的后处理选项。
- 符合 eXpressDSP 软件标准。

#### 4.5.1.4 WMV9 解码器

WMV9 (Windows Media Video 9) 解码器,实现 VC-1 解码标准的 S (Simple) 和 M (Main) 模式,提供高质量的视频流和下载。它支持宽范围的比特率,从 1/2 ~ 1/3 的 MPEG-2 码率,到通过拨号 modem 提供的低比特率的 Internet 视频。

主要性能如下。

- 支持 (MPML, Main Profile Main Level)。

- 支持 VC1 (ML)。
- 接收 RCV 流, 将 ASF 剖析器作为一种应用层。
- 输出 YUV 420/422 格式。
- 支持隔行扫描和平面形式。
- 对 MS 参考解码器, 精确到比特。
- 该解码器不能解码具有以下特性的比特流。
  - Interlaced support in MPML。
  - Sprite mode。
  - X8 intra coding。
  - Beta encoder streams and v8 streams。
- 支持 720p 解码。
- 解码器 API 支持基于帧的解码。

#### 4.5.1.5 MPEG -4 AAC -HE 解码器

MPEG -4 ACC HE 符合 ISO 和 IEC 提出的标准。该 MPEG -4 解码器实现了先进音频编码 (AAC, Advanced Audio Coding) 低复杂度 (LC, Low - Complexity) 以及长期预测 (LTP, longterm prediction) 的目标, 还可以在编译的时候, 选择高效 (HE, high efficiency)、低存储器使用以及低 - MIPS。

该 MPEG -4 ACC 解码器, 是一种宽泛的音频编码算法, 使用了两种基本的编码策略, 极大地降低了高品质数字声所需要的数据量。首先, 去掉感觉不到的以及不影响音频质量的成分。其次, 抑制已经编码的音频信号的冗余成分。通过使用各种感知音频编码和数据压缩工具, 来达到有效的音频压缩。

主要特性如下。

- 实现 MPEG -4 AAC 低复杂度 (LC) 目标类型。
- 在编译进行选择时, 实行 MPEG -4 AAC 低存储器和低 - MIPS profiles。
- 支持 ADIF (Audio Data Interchange Format) 和 ADTS (Audio Data Transport Stream) 输入格式 (所编码的文件符合 ISO/IEC 13818 -7 和 14496 -3 编码器标准)。
- 支持由 ISO/IEC 14496 -3 标准所指定的采样频率, 从 8 kHz 到 96kHz。按照 “AAC\_48KHz” 编译器标志的设置, 该解码器按照 3GPP 标准的要求, 只支持采样频率 8kHz ~ 48kHz。
- 输出 16 - bit 行 PCM 样本。如果产生两个通道的音频数据, 首先连续地存左声道的样本, 然后再存右声道的样本。
- 所支持的最大比特率, 取决于标准所规定的采样频率。

#### 4.5.2 JPEG 图像编解码器

TI 的 JPEG 模块, 符合 ISO 和 ITU 提出的 JPEG 标准。

JPEG 是 Joint Photographic Experts Group 的缩写。该标准是当前最通用的格式, 以 bitmap 格式, 存储高质量的彩色和灰度照片。

JPEG 标准广泛地应用于数码相机和包括因特网在内的其他多媒体应用。充分利用人眼对于

彩色变化的敏感程度不如对于亮度的变化, JPEG 的编码算法, 使系统的设计者通过调整压缩的参数, 来改变图像的损失程度, 从而使设计者得以在文件的大小和图像的质量之间进行折中。

主要特性如下。

编码器:

- 实现对 JPEG 图像的优化实时压缩算法;
- 支持基线的连续编码;
- 支持对采集图像的质量水平的选择;
- 基于每幅图像的模式工作, 可以对图像逐幅编码;
- 支持 YUV 4:2:0、YUV 4:2:2、YUV 4:1:1、YUV 4:4:4 以及灰度输入;
- 在 DaVinci TMS320DM644x 处理器上运行, 不需要协处理器。

解码器:

- 支持基线连续 JPEG;
- 支持 YUV444、YUV422、YUV411 以及灰度彩色子样本格式;
- 支持三个分量的最大值;
- 支持三个量化表的最大值;
- 对于每个 AC 和 DC, 支持四个 Huffman 表的最大值;
- 支持连续 JPEG 的任意图像大小;
- 不支持每个样本 12-bit 的源图像;
- 支持 8-bit 和 16-bit 的量化表;
- 支持 422 隔行扫描或逐行扫描的输出格式;
- 符合 eXpressDSP 软件标准。

### 4.5.3 音频编解码器

TI 的音频编解码器主要包括: MP3 音频编解码器、AAC 音频编解码器、WMA 音频编解码器和 MPEG-2 音频编解码器。

#### 4.5.3.1 MP3 解码器

该 MP3 解码器完全符合 ISO 和 IEC 制定的 MP3 标准。

MP3 解码越来越广泛地用于播放存储在压缩的 MP3 文件中的、CD 质量的立体声音乐。MP3 的全名是 MPEG-1 音频第三层标准。该 DaVinci 技术的 MP3 解码器, 所支持的采样率为 8kHz~48kHz。其输出是高质量的、16-bit PCM 音频信号。

主要特性如下。

- MP3 (MPEG-1 Layer 3) 解码器完全符合 ISO/IEC 11172-3 (MPEG-1 audio)、ISO/IEC 13818-3.2 (MPEG-2 audio), 以及 ISO/IEC 11172-4 标准所规定的 MPEG2.5 标准。
- Layer 1 和 Layer 2 解码器, 只符合 ISO/IEC 11172-3 (MPEG-1 audio) 标准。
- 支持 CBR (Constant Bit Rate, 常数比特率) 和 VBR (Variable Bit Rate, 变量比特率) 流。
- MP3 (Layer3) 解码器支持由 ISO/IEC 11172-3、ISO 13818-3.2 以及 MPEG2.5 标准所指定的采样频率, 8kHz~48kHz。

- Layer 1 和 Layer 2 解码器只支持采样频率 32 kHz、44.1 kHz 和 48kHz。
- Layer 1 解码器所支持的比特率，从 32kbps 到 448kbps；Layer 2 解码器所支持的比特率，从 32kbps 到 384kbps。
- 输出 16-bit 行 PCM 样本。如果产生两个通道的音频数据，首先连续地存左声道的样本，然后再存右声道的样本。
- 可以在 TMS320C64x + 仿真器上，由 CCS IDE3.2.8.0 和 CGTools 5.2.0A04174 实现。
- 该解码器不支持自由格式流。

#### 4.5.3.2 WMA9 (Windows Media 9 Series Audio) 解码器

WMA 是微软公司开发的一种音频压缩文件格式，是微软 2002 年发布的 WM 系列中的一种。

大量的消费类产品，包括手持式音乐播放器、便携式 CD 播放器、机顶 DVD 播放器等，都使用 WMA 文件。WMA 文件也可以在 Windows Media Player、Winamp 以及许多其他的媒体播放器上使用。

主要特性如下。

- 支持各种 WMA 版本。
- 变比特率 (VBR, Variable Bit Rate) 模式。
- Class 4 实现。
- 单声道和立体声系统。
- 符合 Microsoft 所定义的测试标准。
- 输出行 16-bit PCM 信号。
- 输出采样率 8 ~ 48kHz，输入比特率 5 ~ 384kbps。
- 如果使能 WMA-STD，则是单声道或双声道；如果使能 WMA\_PRO，则是 7.1 声道。
- 可以在 TMS320C64x + 仿真器上，由 CCS IDE3.2.8.0 和 CGTools 5.2.0A04174 实现。
- 符合 eXpressDSP 软件标准。

#### 4.5.4 G.711 语音编解码器

该 G.711 编码器/解码器 (codec)，符合 ITU 关于数字语音通信的 G.711 标准，是在传统的电话网络中进行 H.323 音频和视频会议所要求的 codec。

DaVinci 技术所使用的 G.711 codec，将 8kHz 采样的、数字化的、线性的 PCM 输入信号，转换成为 8-bit、按 A-律和  $\mu$ -律压缩的信号。 $\mu$ -律 G.711 算法用于北美和日本，A-律算法则用于欧洲和世界的其他地区。

主要特性如下。

- G.711 编码器，将一个 8kHz 采样的 PCM 输入信号 (16-bit)，线性压缩进一个 64kbps 的 A-律或  $\mu$ -律的比特流。
- G.711 解码器，将 64kbps 比特流，扩展为线性的 16-bit PCM 样本 (8kHz)。
- 符合 ITU-T G.711 规范。
- 对 TI 的 DaVinci 技术以及其他基于 TMS320C64x 的 DSP 做了优化。
- 该编码器和解码器具有 C-可调用的界面。
- 重新进入的多通道实现。



- 其实现与 TI 的 eXpressDSP 软件规则兼容。
- 完全可以中断的代码。
- 可以重新定位的表。
- 支持小端操作。
- 有效的擦刮存储器管理，降低对堆栈的要求。
- 在运行时支持数据缓存器和表的重新定位。
- 可以在使用 CCS IDE 3.2 的 DaVinciEVM 上验证。

## 参 考 文 献

- [1] Mike Hannah, Aaron Kofi Aboagye. Implementation of the Double – Precision Complex FFT for the TMS320C54x DSP. Texas Instruments Incorporated, August 1999.
- [2] Texas Instruments. TMS320C55x DSP Library Programmer's Reference. Texas Instruments Incorporated, May 2013.
- [3] Cesar Iovescu. Wavelet Transforms in the TMS320C55x. Texas Instruments Incorporated, January 2002.
- [4] Texas Instruments. TMS320C55x Image/Video Processing Library Programmer's Reference. Texas Instruments Incorporated, January 2004.
- [5] Texas Instruments. TMS320C55x DSP Library Programmer's Reference. Texas Instruments Incorporated, May 2013.
- [6] Texas Instruments. TMS320C62x DSP Library Programmer's Reference. Texas Instruments Incorporated, October 2003.
- [7] Texas Instruments. TMS320C64x DSP Library Programmer's Reference. Texas Instruments Incorporated, October 2003.
- [8] Texas Instruments. TMS320C67x DSP Library Programmer's Reference Guide. Texas Instruments Incorporated, January 2010.
- [9] Anuj Dharia & Rosham Gummattira. Signal Processing Examples Using the TMS320C67x Digital Signal Processing Library (DSPLIB) . Texas Instruments Incorporated, June 2009.
- [10] Texas Instruments. TMS320C54x DSP Library Programmer's Reference. Texas Instruments Incorporated, October 2004.
- [11] Texas Instruments. TMS320C55x DSP Library Programmer's Reference. Texas Instruments Incorporated, May 2013.
- [12] Texas Instruments. TMS320C62x DSP Library Programmer's Reference. Texas Instruments Incorporated, October 2004.
- [13] Texas Instruments. TMS320C64x DSP Library Programmer's Reference. Texas Instruments Incorporated, October 2003.
- [14] Texas Instruments. TMS320C67x DSP Library Programmer's Reference Guide. Texas Instruments Incorporated, January 2010.
- [15] Texas Instruments. TMS320C64x + IQmath Library User's Guide. Texas Instruments Incorporated, December 2008.

## 第5章 软件开发工具

DSP 技术的发展和 DSP 应用的日益广泛，对于 DSP 开发者来说，除了必须熟悉 DSP 本身的结构和技术指标外，大量的时间和精力要花费在熟悉和掌握其开发工具和环境上。设计人员在为实时系统选择处理器时，都极为看重先进的、易于使用的开发环境与工具。

Code Composer Studio (CCS) 是用于德州仪器 (TI) 嵌入式处理器系列的集成开发环境 (IDE)，也是目前使用最为广泛的 DSP 开发软件之一，所有的 TI DSP 都可以在该环境下开发。CCS 以 Eclipse 开源软件框架为基础。Eclipse 软件框架最初作为创建开发工具的开放框架而被开发。Eclipse 为构建软件开发环境提供了出色的软件框架，并且逐渐成为备受众多嵌入式软件供应商青睐的标准框架。CCS 将 Eclipse 软件框架的优点和 TI 先进的嵌入式调试功能相结合，为嵌入式开发人员提供了一个引人注目、功能丰富的开发环境。CCS 可在 Windows 和 Linux PC 上运行。并非所有功能或器件都与 Linux 兼容，详细信息请参见 Linux 主机支持。

德州仪器 (TI) 的 DSP 开发环境具有直观方便的 IDE、丰富的编程支持、强大的多核开发等特点，其工具主要包括以下几个部分：常用的代码生成工具、调试工具、探针工具、图形工具和分析工具；用于编程支持的片级支持库、DSP/BIOS 工具和 XDC 工具等；用于 ARM + DSP 的异构双核开发的 C6Run、C6Accel、C6FLO 工具。本章介绍这些开发工具和软件的使用方法。

### 5.1 DSP 的集成开发环境 CCS

CCS 包含一整套用于开发和调试嵌入式应用的工具。它包含适用于每个 TI 器件系列的编译器、源码编辑器、项目构建环境、调试器、描述器、仿真器、实时操作系统以及多种其他功能。直观的 IDE 提供了单个用户界面，可帮助用户完成应用开发流程的每个步骤。借助于精密的高效工具，用户能够利用熟悉的工具和界面快速上手并将功能添加至他们的应用。目前的 CCSV5/V4，在使用界面上与老版本 CCS3 有较大差别，但其核心内容差别不大。本节主要介绍 CCSV5 的一些基本特点和使用。详细资料，请参考：<http://www.ti.com/tool/ccstudio>。该网页给出了几乎所有的与 CCS 相关的资料链接。

#### 5.1.1 CCS 的下载和安装

在 TI 公司网站 [www.ti.com](http://www.ti.com) 上可下载不同版本的 CCS 安装文件。

以安装 CCSV5 为例，运行下载的安装程序，当运行到如图 5.1 所示时，选择 Custom 选项，可进入手动选择安装通道。

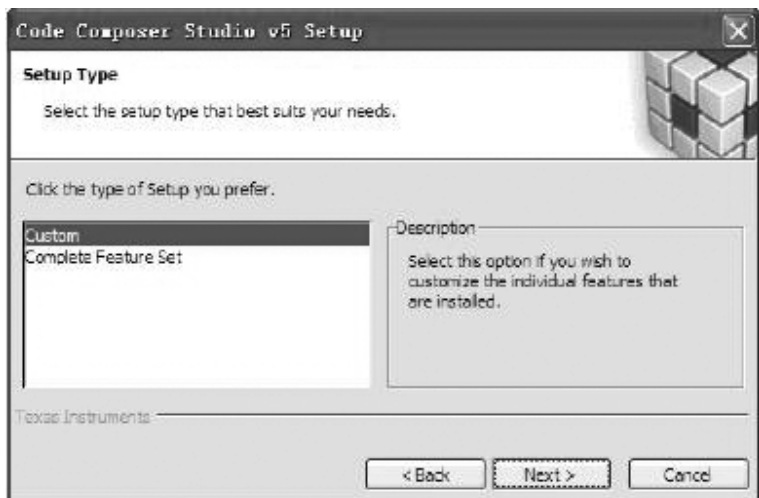


图 5.1 安装过程 1

单击 Next 得到如图 5.2 所示的窗口，为了安装快捷，在此只选择支持 MSP430 Low Power MCUs 的选项。单击 Next，保持默认配置，继续安装。

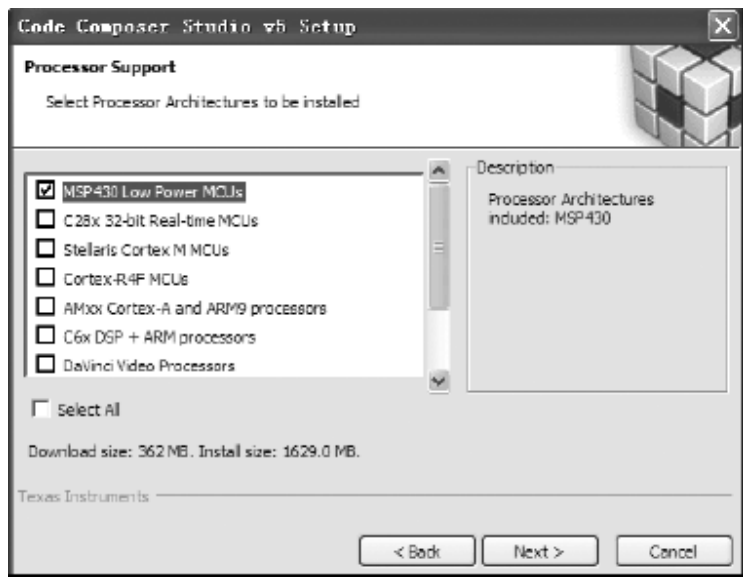


图 5.2 安装过程 2

完成后单击 Finish 将运行 CCS，弹出如图 5.3 所示的窗口，打开我的电脑，在某一磁盘下，创建以下文件夹路径：- \MSP - EXP430F5529\Workspace，单击 Browse，将工作区间链接到所建文件夹。

单击 OK，第一次运行 CCS 需进行软件许可的选择，如图 5.4 所示。在此，选择 CODE SIZE LIMITED (MSP430) 选项，在该选项下，对于 MSP430，CCS 免费开放 16KB 的程序空间，单击 Finish 即可进入 CCSv5.1 软件开发集成环境。

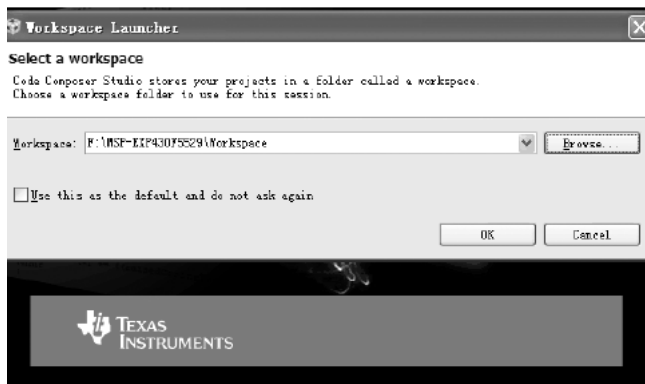


图 5.3 安装过程 3

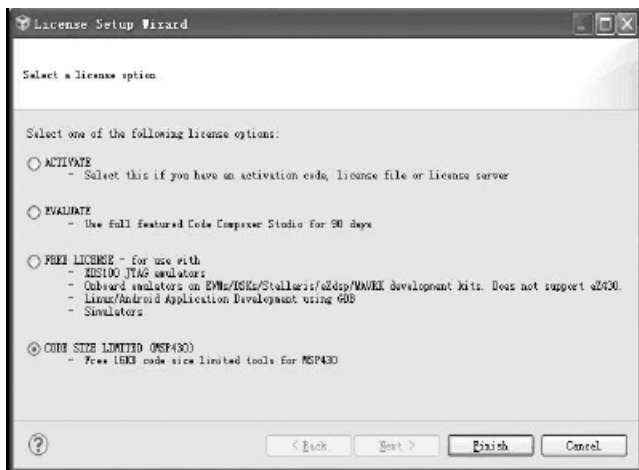


图 5.4 安装过程 4

### 5.1.2 CCS 开发 DSP 程序流程

TI 的 TMS320 系列 DSP 的软件开发一般流程为：先使用汇编语言或 C 语言来编写源程序，然后通过编译、汇编和连接工具，产生 DSP 的执行代码。在调试阶段，可以利用软仿真器（Simulator）在计算机上仿真运行；也可以利用硬件调试工具（XDS510 或 XDS560 等）将代码下载到 DSP 中，并通过计算机监控、调试、运行该程序。当调试完成后，可以将该程序代码固化到 EPROM 或 FLASH 中，以便 DSP 目标系统脱离调试计算机单独运行。

#### 5.1.2.1 CCS 的功能

CCS 包括许多功能，给设计人员提供了极大的方便。CCS 主要包括以下一些功能。

- 集成可视化代码编辑界面，可以直接编写 C、汇编、.H 文件、.cmd 文件等。
- 集成代码生成工具，包括汇编器、优化 C 编译器、连接器等。
- 基本调试工具，如装入执行代码（.out 文件），查看寄存器、存储器、反汇编、变量窗口等，支持 C 源代码级调试。
- 支持多 DSP 调试。

- 断点工具，包括硬件断点、数据空间读/写断点、条件断点（使用 GEL 编写表达式）等。
- 探针工具（probe points），用于算法仿真、数据监视等。
- 分析工具（profile points），用于评估代码执行所需要的时钟周期数。
- 数据的图形显示工具，可绘制时域/频域波形、眼图、星座图、图像等，并可自动刷新（使用 Animate 命令运行）。
- 提供 GEL 工具，用户可以编写自己的控制面板/菜单，方便直观地修改变量、配置参数等。
- 支持 RTDX（Real Time Data eXchange）技术，可以在不中断目标系统运行的情况下，实现 DSP 与其他应用程序（OLE）的数据交换。
- 开放式的 plug-ins 技术，支持其他第三方的 ActiveX 插件，支持包括软仿真在内的各种仿真器（只需安装相应的驱动程序）。
- 提供 DSP/BIOS、SYS/BIOS 工具，增强对代码的实时分析能力，如分析代码执行的效率、调度程序执行的优先级、方便管理或使用系统资源（代码/数据占用空间、中断服务程序的调用、定时器使用等），从而减小开发人员对硬件资源熟悉程度的依赖性。

### 5.1.2.2 利用 CCS 开发 DSP 的流程

在 CCS 环境中，可以打开或新建项目文件（project），用 C 语言或汇编语言编写 DSP 源程序。在 CCS 下开发 DSP 程序，要先建立一个项目文件，然后将源程序文件（C 或 ASM 文件）添加到项目文件中。若用 C 语言开发，在新建项目时，需添加 C 的标准支持库，如对于 C55x 有 rts55、rts55h.lib 及 rts55x.lib 等。此外，还需要将一个内存定位的 CMD 文件添加到项目文件中。若利用 DSP/BIOS 工具开发，还需要一个 DSP/BIOS 的 tcf 文件。有关 DSP/BIOS 工具的使用，在后面的章节介绍。如果要使用 XDC 工具，还需要添加一个 cfg 配置文件。建立好一个项目文件后，可以利用“build all”命令调用代码生成工具（这些工具也存放在 CCS 安装目录的 ccsv5\tools\compiler 下），完成编译和连接。若编译、连接没有错误，便可以启动 Debug 调试工具，在 Run 菜单下用 Load 命令，将生成的 .out 文件装入 DSP 的片内存储器或外部扩展存储器（一般情况启动 Debug 工具后，CCS 自动装入需要调试的 out 文件），开始调试、分析或统计工作。以上所有步骤，都在 CCS 环境下完成。

下面用一个实际的例子，来介绍如何在 CCS 下开发 DSP 程序，在 CCSv5 下调试与运行。该例子虽然是针对 C5500 系列的，对于其他系列，如 C6000，仍然有参考意义。

#### 1. 在 CCS 中完成目标设备的安装与配置，并完成设备连接。

CCS 支持软仿真器、各种型号硬仿真器、各种 DSK 和 EVM。与 CCSv3 不同的是，CCSv4 与 v5 在安装完毕后，将设备接到 CCS 时，需要创建一个后缀为 cexml 的目标板配置文件，来说明 CCS 连接的目标板的类型，以便启动 Debug 调试工具。建立此配置文件后，才能连接设备、加载 .out 文件以及运行 debug 等，简单流程如下。

- 单击 View 菜单下的 Target Configurations，在弹出窗口右上角处，单击 New 按钮新建配置文件。输入文件名及保存位置后单击 Finish，如图 5.5 所示。



图 5.5 新建目标板配置文件

- 在弹出窗口中选择相应设备，完成后单击 Save，如图 5.6 所示。用户可以选择是为单个项目（project）建立目标板配置文件，还是在“User Defined”标签下建立公共的（CCS 下所有项目文件都可使用）目标板配置文件。用户可以建立多个目标板配置文件。

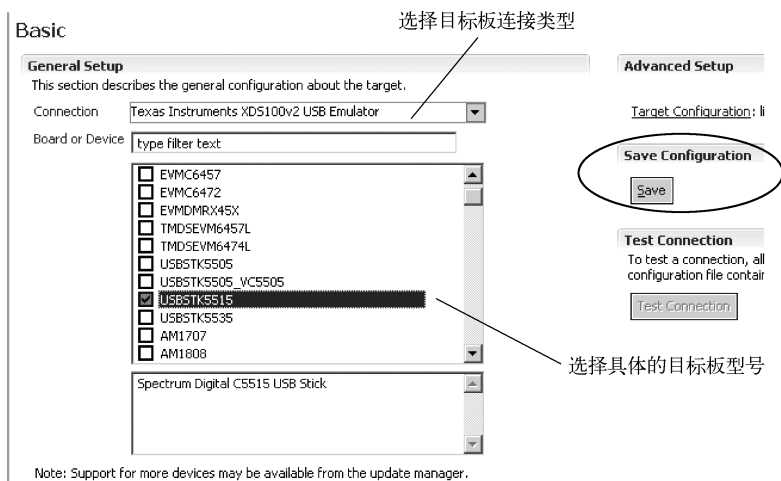


图 5.6 选择目标板的类型参数窗口

- 在 Target Configurations 窗口中找到要使用的目标板配置文件，并右击该配置文件，在弹出菜单中单击 Launch Selected Configuration，启动 Debug 调试工具。在弹出的 Debug 窗口中，通过单击 Run 菜单下的“Connect Target”，完成目标设备与 CCS 的连接。也可以通过右击 Debug 窗口，再选择“Connect Target”。如果要自己添加 GEL 文件，可以在该弹出菜单中选择“Open GEL Files View”，打开 GEL 窗口。CCS 会连接设备并运行 GEL 文件成功后，在 console window 中会显示“Target Connection Complete”。

## 2. 在 CCS 中创建 DSP 项目文件（project）

**例 5.1** 本例为 TI 提供的 TMS320VC5505 eZdsp 的 LED 灯闪烁的示例程序。该例程文件可以在配套光盘（或 spectrumdigital 的网站下载）的 usbstk5505\tests\led 找到。启动 CCSv5，在“File”菜单下的“new”中单击“other”，找到“CCS Project”，选中并单击 Next，开始创立一个新的项目文件，见图 5.7。然后在图 5.8 中，选择对应的设备类型、RTS 库文件（runtime support library）等，单击 Finish 完成 CCS 项目文件创建。



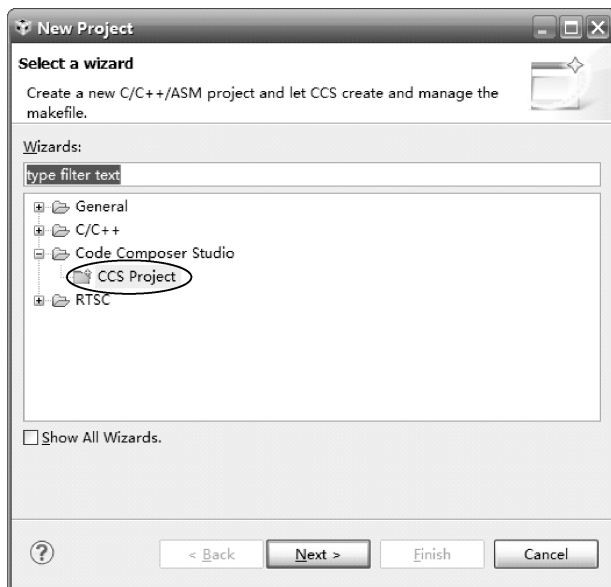


图 5.7 新建 CCS 项目

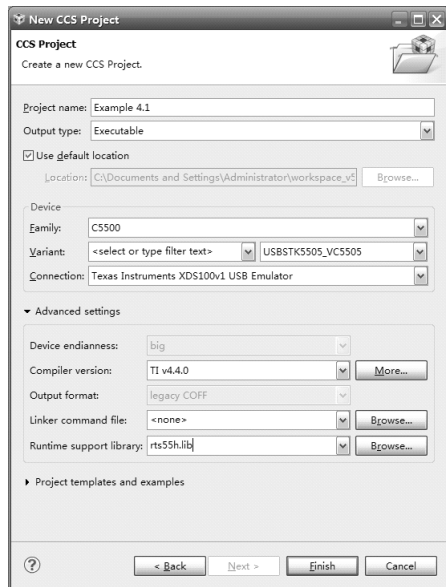


图 5.8 CCS 项目属性参数窗口

1) 单击“view”菜单项下的“Project Explorer”，可查看工作区里的项目文件，如图 5.9 所示。单击每个项目前的“+”可以展开该项目文件。其中 Binaries 为编译链接后最终生成的 .out 文件，而 Includes 文件夹下包含了 C 语言相关的一系列头文件。可右击项目名，利用“Add Files To Project”，将编写好的 main.c 和 led\_test.c 添加到 project 下。

当源文件添加到项目中后，CCS 将自动进行语法分析，且文件中的变量、声明及定义的函数、包含的头文件等都将显示出来，如图 5.10 所示。

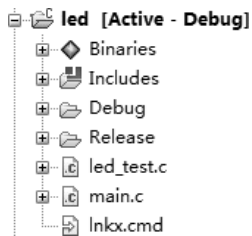


图 5.9 工作区里的项目文件

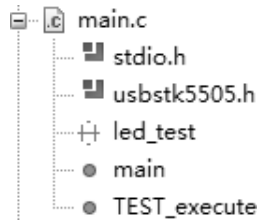
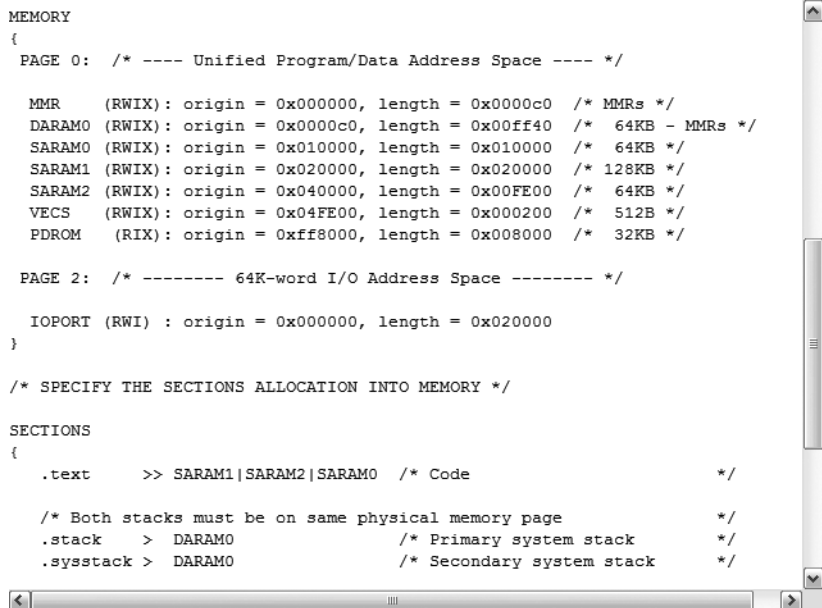


图 5.10 源文件下显示函数与变量

另外一种向 CCS 项目文件（project）中添加文件代码的方式为直接将需要添加的文件拖放或复制到 CCS 项目文件所在目录中即可，CCS 会自动将项目文件所在目录中的所有文件，自动添加到项目文件中。

2) 将内存定位的 lnkx.cmd 文件添加到 project 下，并根据目标系统的实际存储器 and 具体要求，做相应的修改。这里，只使用 VC5505 的片内存储器，SYSTEM MEMORY MAP 有两部分：首先是程序及数据空间，其地址范围从 0x000000 到 0xfffff，其中 MMR、DARAM、SARAM、VECS、PDR0M 的地址表根据 datasheet 做相应的声明。另一部分为 64K 字的 I/O 空间。然后，将源程序中定义的程序和数据段，依次分配到各个存储空间（参见图 5.11）。有关 CMD 文件的具体内容，可以参考本章 CMD 内存定位文件的使用。



```

MEMORY
{
    PAGE 0: /* ---- Unified Program/Data Address Space ---- */

    MMR      (RWIX): origin = 0x000000, length = 0x0000c0 /* MMRs */
    DARAM0   (RWIX): origin = 0x0000c0, length = 0x00ff40 /* 64KB - MMRs */
    SARAM0   (RWIX): origin = 0x010000, length = 0x010000 /* 64KB */
    SARAM1   (RWIX): origin = 0x020000, length = 0x020000 /* 128KB */
    SARAM2   (RWIX): origin = 0x040000, length = 0x00FE00 /* 64KB */
    VECS     (RWIX): origin = 0x04FE00, length = 0x000200 /* 512B */
    PDROM    (RIX): origin = 0xff8000, length = 0x008000 /* 32KB */

    PAGE 2: /* ----- 64K-word I/O Address Space ----- */

    IOPORT   (RWI) : origin = 0x000000, length = 0x020000
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .text    >> SARAM1|SARAM2|SARAM0 /* Code */

    /* Both stacks must be on same physical memory page */
    .stack   > DARAM0 /* Primary system stack */
    .sysstack > DARAM0 /* Secondary system stack */
}


```

图 5.11 CMD 内存定位文件

3) 在“project”菜单项中,选“Properties”,进入编译、连接参数开关选择窗口。例如,重新定义输出文件名,增加优化选择选项等。右击“project”文件夹下的项目文件名,也可进入。

4) 使用“Project”菜单下的“Build Project”或“Rebuild all”命令,完成该项目的编译连接,并生成 DSP 的可执行 OUT 文件。

5) 若编译连接没有错误,就可以启动 Debug 调试窗口。启动并成功连接 (Connect Target) 后,可以利用“Run”菜单中的“Load Program”,将生成的 .out 文件装入目标板。要注意选择生成的 OUT 文件的路径和文件名。特别需要注意的是,CCS 不支持中文路径!当程序被成功装入后,CCS 将自动寻找并打开源代码。PC 指针被设置为 main() 函数开始处,如图 5.12 的调试汇编窗口所示。

6) 很多情况下,为了方便调试,可以在每个项目文件 (project) 中都建立各自的目标板配置文件,或将一个建好的 ccxml 目标配置文件添加到项目中。这样一来,可以直接单击图标,或单击 Run 菜单下的 Debug 选项 (F11) 来启动调试窗口。该命令首先完成编译、连接,然后启动 Debug 窗口,连接设备,并装入 out 文件。如果中间有任何一个错误,都将在“Console”信息输出窗口提示。如果没有错误,也可以进入如图 5.12 所示的调试窗口界面。

7) 在“Run”菜单项中,选择启动程序运行。例如,使用“Go Main”命令,执行到 main() 处暂停;使用“Run”命令,连续运行程序;使用“Step Into” (F5) 或“Step Over” (F6),单步运行程序;……等等,这些操作也可以在 Debug 窗口的工具栏中找到。使用“View”菜单下“Memory”窗口,查看存储器的值;“Registers”查看 DSP 的寄存器;“Expression”观察变量的值。CCS 的一些详细使用操作命令,可参考 CCSV5 的使用手册或 HELP 文档。

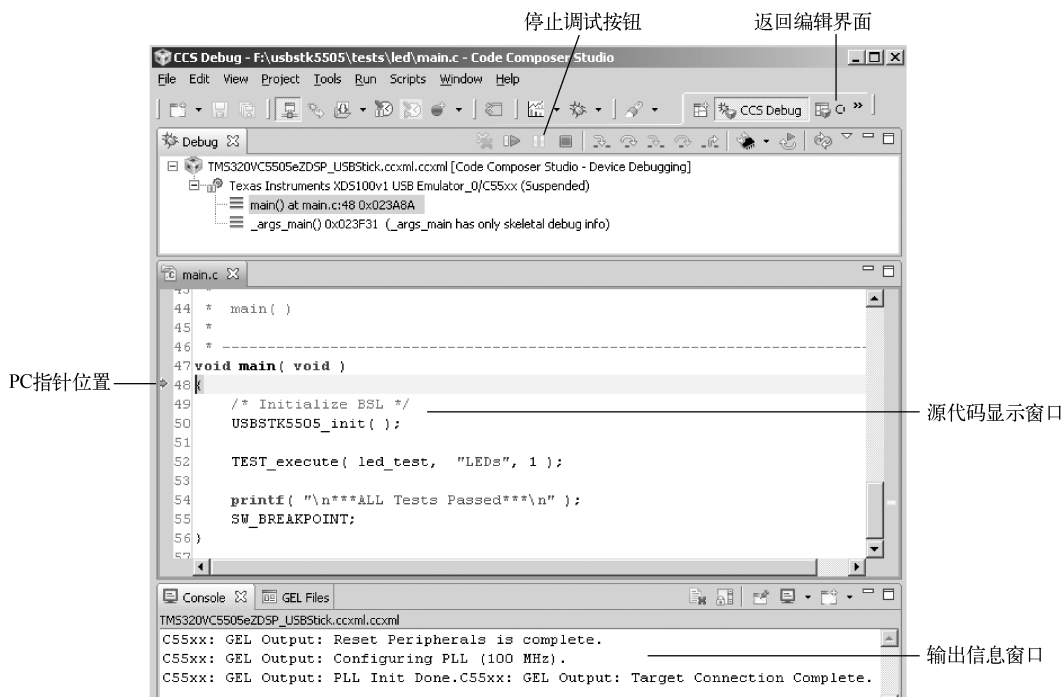


图 5.12 LED 例程 Debug 调试窗口

## 5.2 CCS IDE 常用工具的使用

CCS 提供了集成开发环境,使整个 DSP 软件开发流程都可以在统一界面中完成。对用户而言,熟悉的工具和界面会降低学习难度,更快地上手。在可视化代码编辑界面(Editor),用户可以方便地编写 C、汇编、.H 文件、.cmd 文件等。使用内建的工程/项目管理器(Project Manager),可以很容易管理多用户和多种类项目。本节主要介绍 CCS 中代码生成工具、中调试工具、探针工具、图形工具的使用和分析工具的使用。

### 5.2.1 CCS 中代码生成工具的使用

当选择了一款 DSP,并完成了硬件电路的设计后,便进入 DSP 的软件开发阶段。这个阶段的工作量,往往要占整个系统开发的 70% 以上。通过图 5.13 看出,可以使用汇编语言或 C 语言(Ver 2.0 以上的 CCS 中代码生成工具可以支持 C++)来编写源程序。编写完成后,使用代码生成工具进行编译、汇编、连接,最终形成机器代码。

由于 TI 的各个 DSP 系列所使用的汇编伪指令、机器代码各不相同,所以应使用相应的代码生成工具。

- C 编译器(C compiler)。将 C 源程序编译成为 TMS320 系列对应 DSP 的汇编语言源代码。编译包中包括 C 编译器、优化器(optimizer)以及内部列表公用程序(interlist utility)。优化器可以通过对代码的优化来提高 C 程序的效率。内部列表公用程序将 C 源程序同汇编语言程序结合起来输出,供用户参考,或完成手工优化。

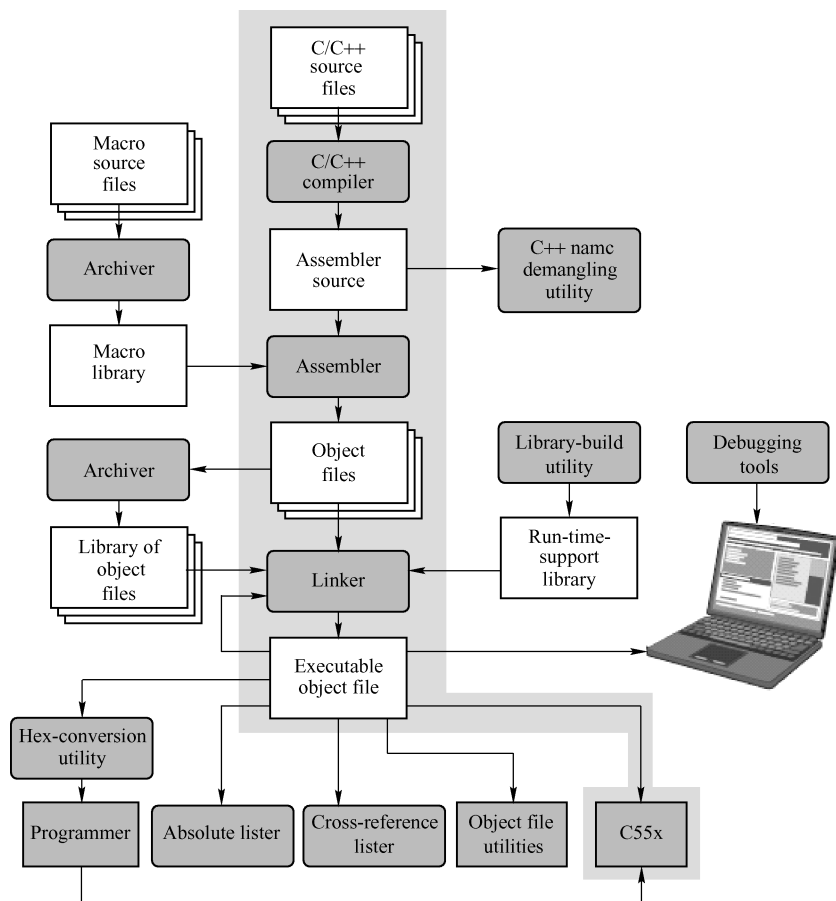


图 5.13 TI 的 TMS320 系列 DSP 软件开发流程图

- 汇编器 (assembler)。将汇编语言源文件转变为基于公用目标文件格式 (COFF) 的机器语言目标文件, 即通常指的 .OBJ 文件。源文件可以包括汇编语言指令 (instruction)、汇编伪指令 (assembler directives) 和宏指令 (macro directives)。C5000 系列提供两种指令集, 用户可以选择使用助记符指令集 (Mnemonic Instruction Set) 或代数指令集 (Algebraic Instruction Set), 但两种不能混用。
  - 连接器 (linker)。将目标文件连接起来产生一个可执行模块。它能调整并解决外部符号的引用。连接器的输入, 是可以重新定位的 COFF 目标文件和目标库文件。
- 以上三个工具是 DSP 软件开发所必备的, 下面的工具是可以选用的。
- 运行支持库程序 (runtime - support utility)。建立用户的 C 语言运行支持库。标准的运行支持库函数, 在 rts.src 里提供源代码, 在 rts.lib 里提供目标代码。若使用 C 语言开发, 应该在连接器工具调用时, 添加该库文件。
  - 运行支持库 (runtime - support library)。包含 ANSI 标准 C 运行支持函数、编译器公用程序函数、浮点算术函数和各个系列 DSP 编译器支持的 C 输入/输出函数。
  - TI DSP 的转换工具。将可执行的 COFF (Common object file format) 文件作为输入, 转化为 TI - Tagged、ASCII - hex、Intel、Motorola - S 或 Tektronix 等目标文件格式, 从而可以将转化文件装载在可擦除程序存储器里。同时, 该转换工具还可以自动生成 BOOTLOADER 程序所需要的引导信息。这类工具是烧写用于 BOOT 的 DSP 代码的必

备工具。TI 的各个系列都有对应的应用笔记（Application Notes）来介绍它们的使用。

类似于其他集成编译环境，当单击“build”按钮时，CCS 会按照事先配置好的选项自动运行编译链接等一系列步骤，生成可执行的目标文件，然后便可加载到 DSP 的内存中调试、运行。

默认情况下，CCS 提供了两套 build 配置选项，分别为 Debug 与 Release。用户可以新建配置，也可以对事先定义的 build 选项进行修改。图 5.14 给出了 CCS 下编译、汇编、连接选项的设置窗口。用户可以右击项目文件名，然后在弹出菜单中选择“Properties”即可。在这个参数选择对话框中，将主要通过选择复选框来进行编译或链接参数的设置，每个选项后面括号中为参数的代码（但不能像之前版本通过输入参数代码来方便地进行设置）。在下面的讨论中，将分别给出一些常用的参数并介绍在 CCSv5 中如何配置，更加详细的参数配置，可以在上述 datasheet 中找到。

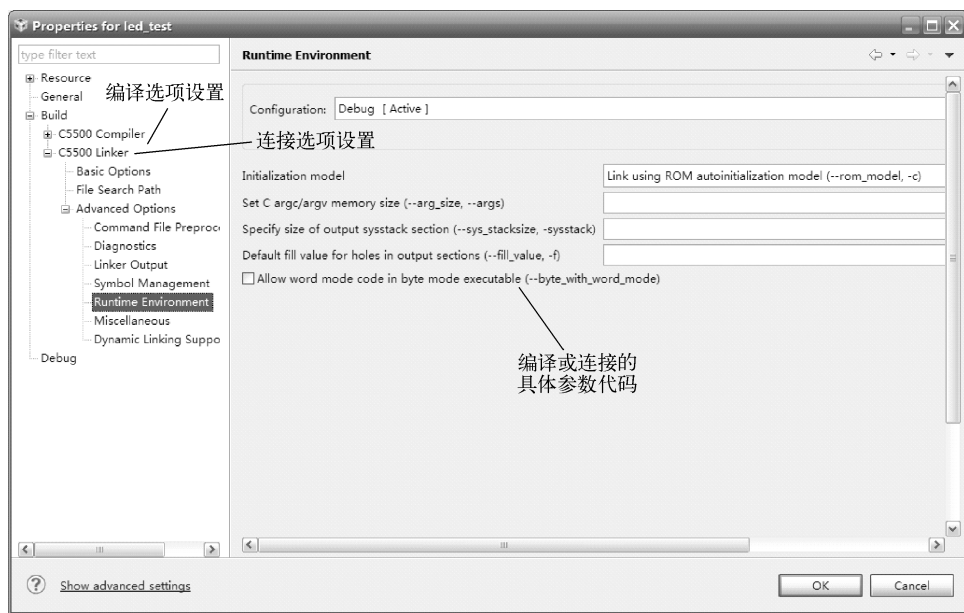


图 5.14 CCSv5 中的 Build 设置

此外，各个工具也可以通过命令行代码单独调用。使用命令行可调用更多的高级功能，不过对于一般开发而言，仅需掌握基本配置，因此这里将不再介绍命令行操作，其详细方法可以在相关用户手册中找到。例如，对于 C55x 系列，可参见 TI 提供的 TMS320C55x Optimizing C/C++ Compiler User's Guide。本小节后面的例子都是 C55x 系列，但对于其他系列，如 C6000 仍然适用。

### 1. 汇编器

TMS320 汇编器，将汇编语言源文件翻译成机器语言目标文件。源文件可以包括汇编语言指令（instruction）、汇编伪指令（assembler directives）和宏指令（macro directives）。汇编伪指令控制汇编过程的许多方面，比如源列表格式、符号定义以及将源代码放入块的方式等。

汇编器的输入文件为汇编语言源文件，其默认的文件扩展名是“.asm”。由汇编器所建立的目标文件，其默认的文件扩展名是“.obj”。用汇编器可以建立列表文件，其默认的文



件扩展名是“.lst”。

汇编器包括以下功能：

- 处理汇编语言源文件中的源语句，产生一个可重新定位的目标文件；
- 根据要求，产生一个源列表文件，并提供对该列表的控制；
- 根据要求，将交叉引用列表添加到源程序列表中；
- 将代码分段；
- 为每个目标代码块设置一个段程序计数器（SPC，Section Program Counter）；
- 定义和引用全局符号；
- 汇编条件块；
- 支持宏调用，并允许在程序内或在库中定义宏。

可以在 CCS 中对汇编器进行配置。以 C55x 为例，配置项位于 C5500 compiler 下的 assembler option 中，如图 5.15 所示。其中常用选项如下。

Figure 5.15 shows the CCS Assembler Options dialog box. The options are as follows:

- ☐ Keep the generated assembly language (.asm) file (--keep\_asm, -k)
- Source interlist: Generate C source interlisted assembly file (--c\_src\_interlist, -ss)
- ☐ Generate listing file (--asm\_listing, -al)
- ☐ Keep local symbols in output file (--output\_all\_syms, -as)
- ☐ Do not generate .clink for .const sections (--no\_const\_clink)
- Simulate source 'copy filename' (--copy\_file, -ahc)
- ☐ Symbol names are not case-significant (--syms\_ignore\_case, -ac)
- Undefine assembly symbol NAME (--asm\_undefine, -au)
- ☐ Do not require '#' on shift counts (mnem only) (--hash\_optional\_shift\_count, -ats)
- Pre-define assembly symbol NAME (--asm\_define, -ad)
- ☐ Suppress all assembler warnings (--suppress\_asm\_warnings, -atw)
- ☐ Generate first-level assembly include file list (--asm\_includes, -api)
- Generate assembly dependency information (--asm\_dependency, -apd) [Browse...]
- Suppress identified assembler remark (--suppress\_remark, -ar)
- Simulate source 'include filename' (--include\_file, -ahi)
- ☐ Generate cross reference file (--cross\_reference, -ax)

图 5.15 CCS 中汇编器的设置

- Symbol names are not case – significant (–ac)：汇编器忽略字母的大小写。例如，abc 与 ABC 是一样的。系统默认要区分大小写。
- Simulate source ‘. include filename’ (–ahi)：设置搜索路径。通知汇编器在指定的搜索路径中去查找 . include 中的文件。
- Simulate source ‘. copy filename’ (–copy\_file, –ahc)：设置搜索路径。通知汇编器在指定的搜索路径中去查找 . copy 中的文件。
- Source interlist：提供 C 与汇编语言的 interlist 功能，将 C 源程序同汇编语言程序结合起来输出，供用户参考，或完成手工优化。
- Generate listing file (–al)：在汇编时产生列表文件，默认后缀为 .lst。

## 2. 连接器

连接器将目标文件连接成一个可执行目标模块。当产生可执行模块时，执行重定位操作



并解决外部引用的问题。连接器接收由汇编器产生的 COFF 目标文件作为输入，也可以接收归档库文件和已经连接好的文件。连接器指令允许合并目标文件、文件块，把块和标志合并到特定的地址中，以及定义或重新定义全局符号。连接器有以下功能：

- 定义一个与目标系统存储器一致的存储模块；
- 集合目标文件块；
- 在目标系统存储器内，将块重新定位到特定的区域；
- 定义或重新定义全局变量，赋予它们特定的值；
- 重新定位块，赋予它们最后的地址；
- 解决未定义的输入文件之间的外部引用。

使用连接命令，可以将一个或多个 .obj 文件连接为一个 .out 文件。在连接时，可以通过段定位控制命令，将不同的代码、数据写入不同的内存单元。值得注意的是，连接器工具生成的 out 文件不是纯二进制代码文件，而是包含代码、符号表、代码定位信息的复合文件。

下面介绍一些 DSP 连接器的常见设置。

- 定义程序的进入点（-e）

如图 5.16 所示，global\_symbol 须出现在源程序中，用 .global 命令说明。C 代码默认的程序进入点（符号）为：\_c\_int00。

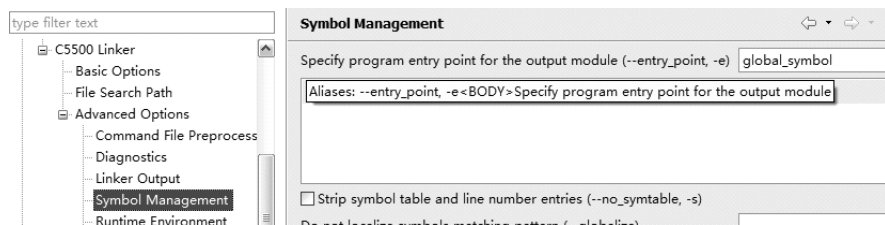


图 5.16 定义程序的进入点

- 初始化模式

在 Runtime Environment 中的 Initialization mode 选项处可找到，两种模式分别介绍如下。

使用 C 编译器的 ROM 初始化模式（-c）：代码中定义的有初始值的变量，将在 main() 函数前由系统提供的初始化代码设置。

使用 C 编译器的 RAM 初始化模式（-cr）：代码中定义的有初始值的变量，在代码装入后就设置好了。

- 指定库文件的路径（-i）

如图 5.17 所示，可通过工具按钮添加、删除、编辑程序中文件库文件的路径。

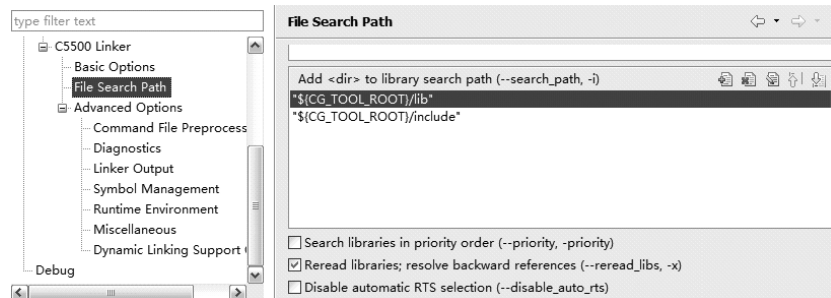


图 5.17 指定库文件的路径

➤ 指定生成 map 文件名 ( - m)

该选项在 Basic Option 中, map 文件将列出所有变量、函数名的符号所对应的地址, 以及内存资源的占用情况。

➤ 指定生成的 out 文件名 ( - o)

该选项在 Basic Option 中, 指定编译、连接后最终生成的 DSP 可执行代码 out 文件的文件名。

### 3. 优化 C 编译器

每个开发商都关注尽可能缩短产品的开发周期和代码维护。TI 为 DSP 开发者提供了一套用于 TMS320 DSP 的优化编译器。该 ANSI C 优化编译器将标准的 ANSI C 语言文件, 转换为高效的 TMS320 汇编语言源文件; 再通过 TMS320 的汇编器和连接器, 生成 DSP 的可执行代码。下面以 C55x 优化 C 编译器为例, 简单介绍优化 C 编译器的基本配置与一些高级配置。

如图 5.18 所示, 基本配置提供了 4 种优化等级, 可在 optimization level ( - - opt level, - O) 中选取。低级别的优化由代码生成工具完成, 高级别的在优化器中完成。选择高等级 (例如 - - opt level = 2 或 3), 可以得到更为优化的代码。

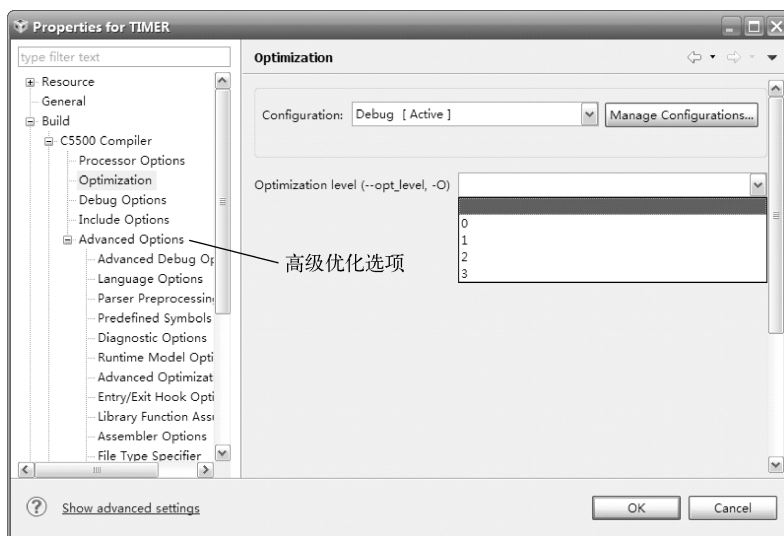


图 5.18 优化 C 编译器基本选项

优化器还提供了其他一些功能, 可在 Advanced Optimization 中找到。

➤ generating an Optimization Information File

生成 level[0 - 2] 的优化信息文件, 0、1、2 分别对应不生成、生成、生成详细文件。

➤ Generating optimized source interlisted assembly ( - - optimizer\_interlist, - os)

优化器输出中将同时列出汇编的注释与汇编源文件。C55x 代码工具的 Interlist 功能就是通过该选项以及上文汇编器中的 source interlist 选项来实现的。

➤ Assume called funcs create hidden aliases ( - - aliased\_variables, - ma)

如下情况时, 可调用该功能进行优化: 存在一些函数, 仅仅是返回输入参量的地址, 或将其赋予全局指针。

更详细的内容, 可参考各个系列 DSP 的优化 C 编译器的相关手册资料。

### 5.2.2 CCS 中调试工具的使用

程序的调试过程和优化过程,一般会占用整个开发周期的 60% 以上的时间。好的调试工具可以大大提高调试的效率。调试中往往需要很多技巧和手段,但都必须要有良好的工具才能实现。针对 DSP 的特殊应用环境,CCS 除了提供大多数通用开发环境都具备的基本调试工具,如存储器与寄存器的查看与修改、断点、性能分析外,还提供在嵌入式开发中非常有用的事件检测、探针等工具,以及在信号处理类应用中非常实用的图形化工具。合理有效地使用这些工具,可以极大地提高程序调试的效率。下面的例子是在 C55x 系列下完成的,对于 C6000 等其他系列,这些命令或操作同样适用,只是显示内容可能会不同。

#### 1. 装入并运行应用程序

在建立项目文件,完成程序的编辑、编译和连接后,得到可以在目标系统中运行的可执行程序 .out 文件。为了调试应用程序,首先必须将程序加载到目标系统中。前面的 5.1.2 小节已经介绍过相关连接目标板并装入 OUT 代码的过程。正如图 5.12 所示,在 CCSv5 中,启动 Debug 工具窗口并装入 OUT 文件后,大多数情况会自动指向 main() 函数处。通过源代码窗口,可以看到代码逐行运行的情况。但有时需要查看汇编代码。通过 View→Disassembly 来打开汇编代码窗口,如图 5.19 所示。如果需要,可以单击汇编窗口 (Disassembly) 的工具栏图标 (Show Source 按钮),来切换到 C 和汇编混合模式显示。

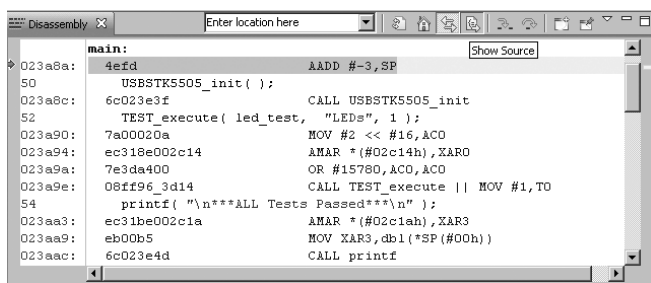


图 5.19 反汇编窗口

若选择菜单 Run→Resume (F8), 程序开始自由运行; 若选择菜单 Run→Halt, 中断程序的运行, 当再次选择 Resume, 程序从当前位置继续运行。在 Run→Load 下拉菜单中, 还有几个与装载程序相关的菜单。

- Reload Program 重新加载程序。对源程序做了修改, 重新汇编、连接后, 需要将新生成的可执行程序加载到目标系统, 可以使用此菜单。
- Load Symbol 加载符号。可执行程序中一般会有一个符号表。调试过程中, 特别是高级语言程序的调试过程中, 符号表非常重要。在符号表中, 包含了程序中使用到的各种符号, 如变量名、函数名等在数据存储器 and 程序存储器中对应的地址。在程序调试过程中, 开发人员需要查看和修改这些变量的值时, 不需要知道它们具体的地址, 只需给出变量名, CCS 会从内部符号表中得到对应的地址。一般加载程序时, 自动将 .out 文件中的符号信息加载到内部符号表中; 也可以利用这个菜单只将生成的 .out 文件中的符号信息, 加载到 CCS 内部的符号表中。装入符号表以便看到汇编代码所对应的 C 源代码, 这样的调试方法, 在 linux 的 U-BOOT 调试中非常有效。
- Add Symbol 增加符号。将指定的 .out 文件中的符号信息, 添加到 CCS 内部的符号表

中。和加载符号不同，加载符号会清除 CCS 内部原来的符号表。

在 Run 菜单下，与程序运行相关的其他菜单项。

- Step Into: 单步运行程序：如果在 C 源程序窗口，则执行一条 C 语句；如果在反汇编窗口，则执行一条汇编语句；如果碰到子程序调用，则进入子程序执行。
- Step Over: 单步运行程序，将子程序调用看成一条语句。
- Step Out: 单步执行到所在的子程序结束处。
- Restart: 重新运行程序。
- Run to line: 执行到光标所在的程序行，注意，光标可能会处于永远也运行不到的位置。
- Reset CPU: CPU 复位，进入 CPU 的复位中断。

## 2. 存储器/变量的查看与修改

调试工具的最基本功能之一，是查看和修改 DSP 上的寄存器、程序存储器、数据存储器、IO 存储器，以及以高级语言中定义的数据结构查看和修改变量。CCS 提供了非常灵活的方式实现这些功能。

### 1) CPU 寄存器

用 Halt 停止程序运行后，选择菜单 View→Registers，在工作区打开一个 CPU 寄存器窗口，其中列出了所有 CPU 内部及外设寄存器的值。选择菜单 Run→Resume（或按 F8）继续程序运行，CPU 寄存器的值肯定会发生改变。但是，通过观察发现，CPU 寄存器查看窗口中的值不会相应地变化。再用 Run→Suspend 停止程序运行，可以发现，窗口中的值发生了变化。其中，与上次相比，发生变化的寄存器以红色显示。在 CCS 中，各种显示，都是在程序运行停止或者碰到断点、探针点时才会刷新。

修改寄存器值的操作非常直观，单击想要修改的寄存器的 value 栏即可对其修改，值得一提的是，在 CCS 中，凡是这种需要输入值的地方，都可以输入一个合法的 C 表达式，即采用常用的 C 运算符将各个常量、CCS 内部符号表中的符号等组合得到，例如图 5.20 中，输入 AC1 + AC2 回车后将得到 0x000000002。用鼠标 + Ctrl 键选定一个或几个寄存器，右键，可以进行复制或在 Format 中选择显示方式，如 16 进制、2 进制、10 进制、float 型等。

### 2) 外设寄存器

在 DSP 芯片内部有各种外设，如定时器、HPI、McBSP 等。对这些外设的控制和运行，都以读写外设控制寄存器的方式来完成，CCS 的调试工具允许查看和修改这些外设寄存器。同样通过菜单 View→Registers，在打开的寄存器窗口中找到对应的外设寄存器菜单并展开，如图 5.21 所示。图中展开了 I2C 外设，并显示了 I2C 外设的各个寄存器的内容。

AC0	AC1+AC2
AC1	0x0000000001
AC2	0x0000000001

图 5.20 修改寄存器的值

Name	Value	Descri
DMA0		
DMA1		
DMA2		
DMA3		
EMIF		
GPIO		
I2C		
IC0AR	0x0000	Memor
IC1MR	0x0000	Memor
ICSTR	0x0410	Memor
ICCLKL	0x0000	Memor
ICCLKH	0x0000	Memor
ICCNT	0x0000	Memor

图 5.21 外设寄存器

对不同的目标 DSP 处理器，外设寄存器的显示差别较大。例如，C5505 的外设寄存器在 CCS5 中显示得非常详细，包括与定时器相关的 TIMER、RTC 等，四个 DMA 的寄存器、I2S 与 I2C 总线的寄存器，以及 USB 与 GPIO 相关的寄存器等。

### 3) 数据查看和修改

选择菜单 View→Memory Browser，出现 Memory 窗口，如图 5.22 所示。通过该窗口可以显示一块存储器的内容。点击该窗口工具栏的 New Memory View 可以新打开一个窗口，同时查看另一块存储器内容。

其中，左上角的复选框为 Page 栏，选择数据来自哪个存储器空间：程序存储器（PROGRAM）、数据存储器（DATA）、还是 IO 空间（IO）。Page 栏一旁的方框为 Address 栏，输入需要显示的存储区的起始地址，此处既可以直接输入地址，也可给出一个合法的表达式。例如，有一个长度为 16K 字的名为 buffer 的缓冲区，需要显示中间部分，可以用 buffer + 8 096。在二者

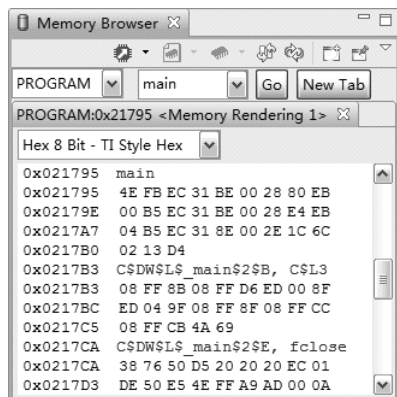


图 5.22 存储器显示

下方的复选框为 Format 栏，选择显示数据的格式。数据以二进制形式存放在存储器中，为了得到直观的结果以方便调试，必须选择合适的显示格式。可以使用的显示格式有：

- Hex m bit - C Style, C 语言风格的 16 进制，以二进制的 8bit 为一组显示；
- Hex m bit - TI Style, TI 风格的十六进制，与 C 语言风格的主要区别是，每个数据前面没有“0x”表明是十六进制；
- 16 - bit Signed Int, 16 - bit 有符号整数；
- 16 - bit Unsigned Int, 16 - bit 无符号整数；
- 32 - bit Signed Int, 32 - bit 有符号整数；
- 32 - bit Unsigned Int, 32 - bit 无符号整数；
- Character, 每个字的低 8 - bit 作为一个字符；
- Packed Char, 每个字的低 8 - bit 和高 8 - bit 各作为一个字符显示；
- 32 - bit Floating Point, 32 - bit 浮点型；
- 32 - bit Floating IEEE Point, 32 - bit IEEE 格式浮点型；
- 32 - bit Exponential Floating Point, 32 - bit 浮点型，以指数方式表示；
- 32 - bit IEEE Exp'l Floating Point, 32 - bit IEEE 格式浮点型，以指数方式表示；
- Binary, 二进制形式显示。

如果需要修改某个存储单元的内容，可以在相应的存储单元上双击，输入值后回车即可。此外，还有几个功能与整块存储器操作有关，在调试过程中经常用到。

- Copy: 用鼠标选定一定区域右击，选择 Copy to memory 菜单中的 current selection 或 visible cells，可以完成从一块存储器到另一块存储器的复制，在出现的对话框中，设置目标存储器的地址。
- Fill: 可以完成对一段存储器填充特定的值。右键显示区，单击 Fill memory，在弹出的 Setup Filling Memory 对话框中，设置填充存储器的地址、长度、页及填充值等信息后，CCS 完成对 DSP 存储器的填充。
- Load/Save: 可将存储器中的数据整体保存为一个文件，或加载数据文件到存储器。



#### 4) 使用数据参考缓冲区

如果希望使用参考缓冲区, 则右键存储器显示窗口, 在 Reference Buffer 菜单中选择 set-up。所谓参考缓冲区, 实际上位于主机上 CCS 的内部缓冲区, 它用作 DSP 中一段存储器块的

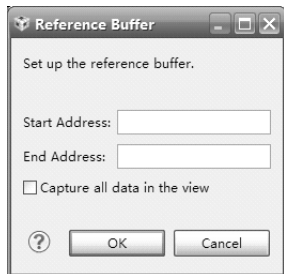


图 5.23 参考缓冲区设置

的镜像。每次需要刷新存储器显示窗口时, 改变的数据单元会以桔红色显示。但是, 向前后滚动显示窗口时, 即使此次刷新时, 这些存储单元的值改变过, 由于没有保存原来的值, 无法比较, 因此不能用红色显示。此时, 如果对 DSP 中一段存储区的数据改变比较感兴趣, 可以使用参考缓冲区。每次刷新时, CCS 会将需要显示的存储器数据与参考缓冲区比较, 改变的单元以红色显示。如果使用了参考缓冲区, 在弹出的如图 5.23 所示的窗口中 Start 和 End 栏指定起始地址和结束地址, 也可单击 capture all data in the view 来自动填写 memory 窗口显示范围中的地址范围。

设置好属性后, 单击 OK, 再在 Reference Buffer 中点击 Eable 和 Capture 开始设定参考, 继续运行, 当值发生改变后就会以红色显示, 光标移动到该位置, 会显示其地址、修改前的值及当前值。

#### 5) 变量的查看和修改

在高级语言程序的调试中, 变量具有特定的数据结构, 而不仅仅是一串二进制数, 按照结构查看数据非常方便。

- 源程序中显示: 将鼠标移动到源程序的某个变量上, 出现一个小的工具提示窗口, 其中有该变量的值, 以便快速查看。
- Variables 窗口: 单击菜单 View→Variables, 在该窗口中可以查看当前局部变量。右击相应变量, 在 Format 中可以更改显示方式, 如 hex (十六进制)、dec (十进制)、bin (二进制)、float (浮点格式) 等。
- Expression 窗口: 在源程序中勾选变量名, 然后右击选择 Add watch expression 可在弹出的 Expression 窗口中显示。也可以单击菜单 View→Expression, 单击 Expression 窗口的空白栏 <new> 处, 输入变量名查看。

### 3. 断点工具的使用

使用断点, 是程序调试的基本手段。在程序调试过程中, 充分利用 CCS 提供的灵活多样的断点工具, 可以大大提高调试效率。CCS 的断点工具, 包括软件断点、硬件断点等。此外, 在 CCS v5 中, Probe 功能将通过断点来实现, 同时 Profile 功能也整合到断点中, 集中到了一起, 方便用户调试。下面将分别介绍断点工具的使用, 并在后面的小节中分别介绍探针和分析工具。

#### 1) 软件断点



软件断点是最常用的断点形式, 即在已经加载到程序存储器中的程序的某一行上, 设置断点。程序运行过程中, 如果碰到断点, 就会暂时停止运行, 回到调试状态。此时, 用户可以通过查看变量、图形等方法, 了解运行的情况, 发现程序中的错误。

在源程序窗口, 将光标移动到需要添加断点的行右击, 在弹出式菜单中选择 new breakpoint→breakpoint。在本行左边会出现蓝色对钩状的断点标记, 表示设置了断点。也可以直接在源程序最左侧状态列 (行计数的左边一列) 空白处双击来设置断点, 再次双击取消断点。重新启动程序运行, 程序会停在断点处, 并且在此行左边显示一个小箭头, 表示程序当



前运行到此,如图 5.24 所示。

程序碰到断点停止后,自动刷新 CPU 寄存器窗口、存储器和变量查看窗口,以及图形显示窗口等。按 F8 键,程序将继续运行。

在程序调试中,会使用到许多断点。其中某些断点可以临时关闭,以后需要使用的时候再打开。如果将暂时不需要的断点都清除,以后添加起来会比较麻烦。CCS 中的断点,具有打开和关闭两种状态。选择菜单 View→Breakpoints,出现 Breakpoints 窗口(如图 5.25 所示)列出了所有的断点。每个断点前面有一个检查框,如果检查框被选中,表示对应的断点处于打开状态;没有选中,则对应的断点处于忽略状态。另外,还可以使用窗口工具栏的按钮,完成一些断点的操作。其中,  (Remove selected breakpoints) 删除列表框里选中的断点;  (skip all breakpoints) 使所有的断点处于忽略状态,等等。

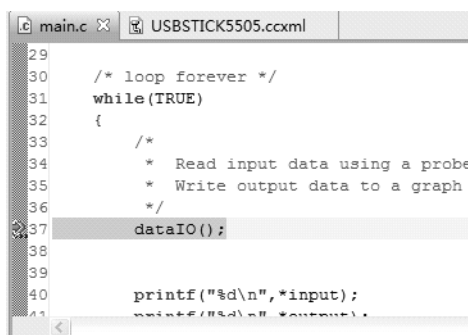


图 5.24 程序停在断点处

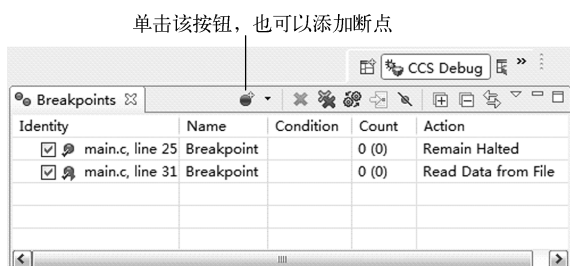


图 5.25 断点管理窗口

前面使用的软件断点,称为无条件断点,即只要程序运行到断点所在的地方就会停止。还可以为每个软件断点附加一个条件,只有条件满足时才停止。例如,断点处的语句在一个循环中,则第一遍循环碰见断点就会停下来。如果希望循环进行 50 遍后才停下来,就可以使用条件断点。

右击图 5.25 中的断点,在菜单中选择 Breakpoint Properties,在弹出对话框(如图 5.26 所示)的 Condition 中,填写表达式,即只有当指定的表达式为“真”时,才在指定位置停止运行。条件表达式必须是一个合法的 C 语言表达式,可以使用 CCS 内部符号表中的符号,如果是局部变量,则断点处局部变量必须可用。

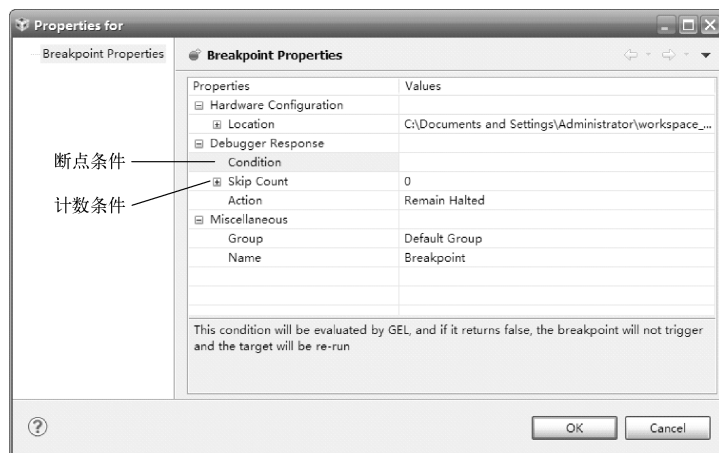


图 5.26 断点属性设置

## 2) 硬件断点的使用

前面讲到的软件断点，是通过修改断点处的指令，来达到中断程序运行的目的。这样的断点用得最多，但功能上还是有一定的限制。例如，断点必须位于程序存储器，而且必须是在可由 CCS 修改的存储器，如 RAM 中。如果需要为 ROM 中的程序设置断点，软件断点就无能为力了，必须使用硬件断点。

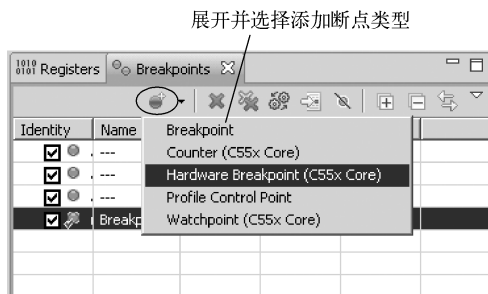


图 5.27 添加硬件断点

在图 5.25 的断点管理窗口下，单击窗口上方的工具栏中的添加按钮旁边的小三角符号，便可以添加断点的类型，其中 Hardware breakpoint，就是硬件断点，如图 5.27 所示。硬件断点同样有一些属性，比如计数器 count，用来设置第几次碰到断点时停止运行。

### 5.2.3 CCS 中探针工具的使用

在 CCS 环境下，可以设置探针。当程序运行到探针位置时，CCS 中断目标系统的 DSP 程序的运行，从与该探针连接的数据文件（存放在 PC 里）中读出数据或输出结果。完成数据的传输后，自动恢复目标系统的 DSP 程序的运行。探针工具特别适用于算法的仿真。在目标系统，特别是输入/输出硬件电路完成之前，可以使用该工具完成数据的模拟，调试软件算法。探针工具可以运行在软仿真（Simulator）下。

探针实际上是一种特殊的断点，在较早版本的 CCS 中，有单独添加及设置 Probe point 的功能；而在新版本中，该功能被添加到了 breakpoint 下，通过设置运行到断点时的 Action，来实现探针的功能。下面结合上一节的例子，介绍探针工具的使用。

**例 5.2** 在图 5.24 中，有一个 dataIO() 空函数，由于没有输入数据的硬件支持，所以该函数没有任何意义。可以在该处使用探针工具，来模拟数据输入。

(1) 装入编译连接后的 out 文件，在 dataIO() 前添加断点。

(2) 在 breakpoint 查看窗口中右击该断点，选择 Breakpoint Properties，可以在弹出对话框中看到 Action 的选项，若选择 Read Data from file，则可设置为探针，表示将从 PC 的一个文件中读取数据，如图 5.28 所示。

(3) 为探针建立连接的数据文件。在“File”栏中选择要使用的数据文件。注意，所使用的数据文件的类型为 \*.dat。CCS 中支持的数据文件有一定的格式规定，文件头须为：

[固定标识(1651)] [数据格式] [基地址] [页类型] [数据长度]

其中，数据格式：1 - 十六进制，2 - 十进制，3 - 十进制长整型，4 - 十进制浮点型。

页类型：0 - 数据，1 - 程序。

对于探针的数据输入来说，数据格式要严格填写，而后面三项并不重要，在 dat 文件中可以省略。例如：十六进制数据格式文件表示为

```
1651 1 0 0 0
0xee4d
0xc000
```

0x08bb

.....

(4) 在“**Start Address**”项中,输入接收数据的内存地址。当装入 OUT 文件后,可以使用符号,如 `inp_buffer`。Page 为地址所在的存储空间,如 DATA、I/O。在“**Length**”项中,输入一次接收数据的长度,如 100。这表示当程序运行到探针点时,从设置的数据文件中读出 100 个数据到 `inp_buffer` 缓冲中。“**Wrap around**”选项意为,每当读取到数据文件的末尾后,下次读取将从文件开头进行。

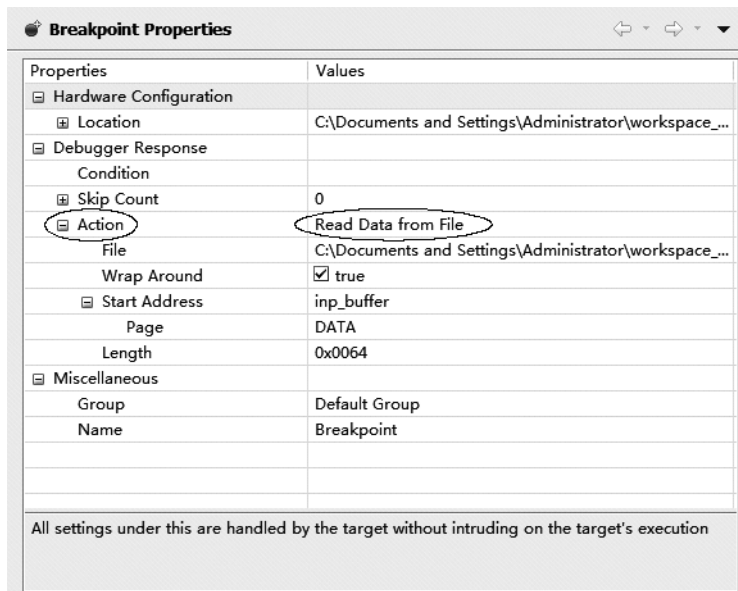


图 5.28 添加探针,并为探针建立相连接的数据文件

(5) 到这里,已经完成了整个探针的设置工作。当程序运行起来后,每运行到 `dataIO()` 函数处,CCS 会自动从 PC `sine.dat` 中读出 100 个数据,放入 `inp_buffer` 数组,从而实现数据的模拟输入。

#### 5.2.4 图形工具的使用

CCS 提供了多种绘图工具,将内存中的数据以各种图形方式显示,帮助用户直观地了解数据的意义。需要注意的是,图形窗口只有在遇到断点时才刷新,所以应增加断点。图形工具既可以显示某时刻、某段数据的图形,也可以显示波形的动态变化。

若读者使用过之前版本的 CCS,可能会熟悉 `Animate` 这一功能。使用 `Run` 命令和 `Animate` 命令运行程序的区别为:使用 `Run` 运行时,CCS 将在断点处停下来,直到再次发出运行命令,程序才能继续运行;而 `Animate` 运行时,CCS 遇到断点后,先暂停程序运行,对打开的变量、寄存器、存储器、图形等窗口进行刷新操作,然后自动恢复程序运行。由于程序持续运行,刷新频率较高,因此图形窗口会显示出动态的效果。

在新版本的 CCS 中,同样保留这一功能,但操作方式改为对断点进行设置:在 `Breakpoint Properties` 对话框的 `Action` 选项中,选择 `Refresh all windows`,即可在到达断点时刷新并

继续运行,从而实现观测动态效果的功能。其实,这样的设计可以实现在某些断点处自动刷新窗口,同时也可在另外一些断点处停下来,使程序的调试更加灵活方便。CCSv5 中将数据波形显示和图像显示分为两个命令,下面分别简单介绍。

### 1. 数据波形显示例子

下面继续前一节的例子,通过图形显示窗口,观察输入输出波形的变化。

**例 5.3** 在例 5.2 中,建立了探针点,从数据文件中读入模拟输入信号。本例使用图形显示工具,通过波形显示的方式,观察输入/输出的结果。

➤ 在“Tools”菜单项中选“Graph”,然后选“Single time”进入图形设置窗口,见图 5.29。

➤ 输入绘制图形的数据起始地址“Start Address”。装入 OUT 文件后,起始地址可以是符号,如 `inp_buffer`。

➤ 输入绘制图形的数据 BUFFER 的长度。在 CCS 的图形工具设置窗口中,有两个参数: Acquisition Buffer Size 和 Display Data Size。前者表示这个数据缓冲区的大小,后一个参数表示 CCS 图形工具绘制波形使用的数据大小,这里都设置为 100。

➤ 在“DSP Data Type”栏中,设置相应的数据类型,这里选择“16-bit signed integer”(16-bit 带符号整数)。

➤ 在 `dataIO()` 函数的调用处设置一个断点,一种方法为将光标移动到该行右击,在弹出菜单中选择“Breakpoint”。按 F8,连续运行程序,程序将在断点行(`dataIO()` 函数行)暂停。此时,波形显示窗口将出现输入信号的波形,这是从探针点得到的数据波形。这里需要特别说明的是,这里添加断点,主要是因为图形工具在断点处刷新数据,更新显示。从而看到输入数据的波形,如图 5.30 所示。

➤ 再打开一个波形显示窗口,将“Start Address”改为输出缓冲 `out_buffer`,显示输出波形。

➤ 将断点的 action 设置为 refresh all window 运行程序,可以看到连续的输入/输出波形。

此外,CCS 的图形工具可显示其他多种波形,可以通过 Tools→Graph 菜单下选择,如 FFT 波形、功率谱波形、眼图等。通过“Autoscale”,选择是否使用自动刻度,若不使用,可以设置显示波形的最大、最小值。CCS 的图形工具还可以通过设置“Display Data Type”,识别不同的数据类型,如无符号整型、有符号整型、浮点型等。通过设置“Sampling Rate”,使得频谱波形的值与实际的原始信号相对应。

### 2. 图像数据显示例子

在 Debug 窗口的 Tools 菜单下,可以打开 Image Analyzer 工具来显示图像。通过图像显示窗口的 Import Properties,可以设置或导入显示图像所需要的参数。图 5.31 给出了显示一个  $256 \times 256$  的 8-bit 灰度黑白图像的参数设置情况。

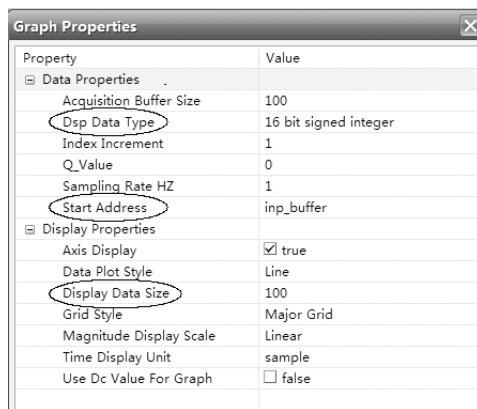


图 5.29 设置图形显示的参数

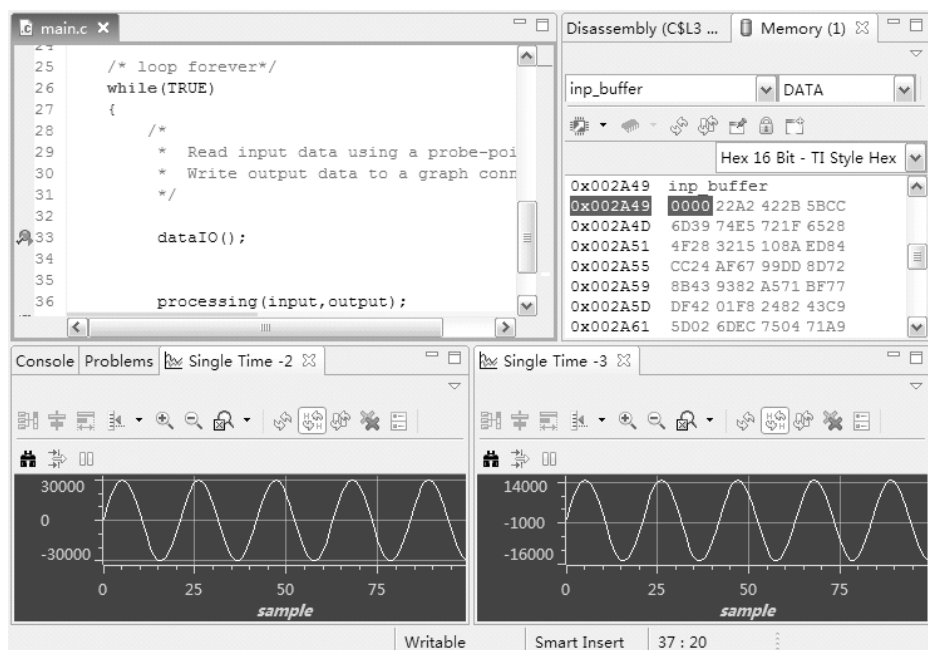


图 5.30 利用探针、图形工具仿真算法

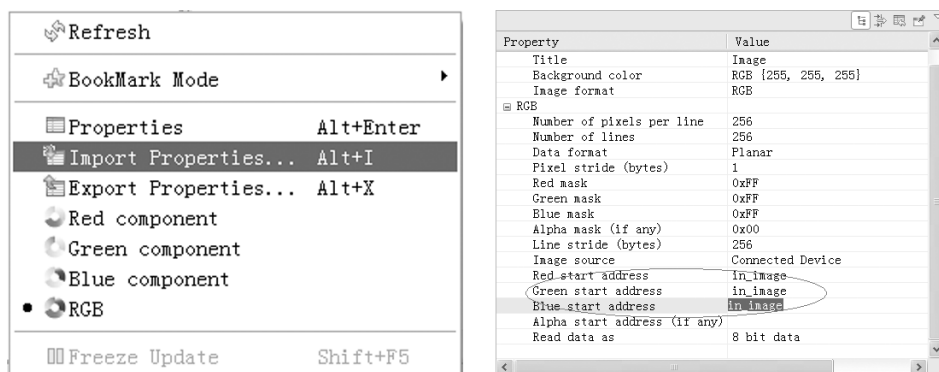


图 5.31 显示 256 × 256 的灰度图像参数配置

**注意：**图 5.31 中的 start address 就是图像数据的地址！这里按 GRB 格式显示，需要分别给出红、绿、蓝三个分量的数据地址。由于例子输入图像是黑白图像，所以 RGB 的地址值都一样。

### 5.2.5 分析工具的使用

在 CCS 中，可以利用代码分析工具，计算代码执行了多少个机器周期。CCS1.2 提供的分析工具比较简单，在 CCS2 中，该工具的性能有了很大的改善。在 CCS2 和 CCS3 中，可以单独分析一个函数或一段代码，使用起来也比 CCS1.2 方便，在统计结果的显示方面，增加了分析区域内函数调用与否的分类统计结果。而 CCS4 和 CCS5 两个版本中，分析某段函数和代码将通过设定 Profile Control Point 来完成。下面结合实际例子来介绍分析工具的使用。



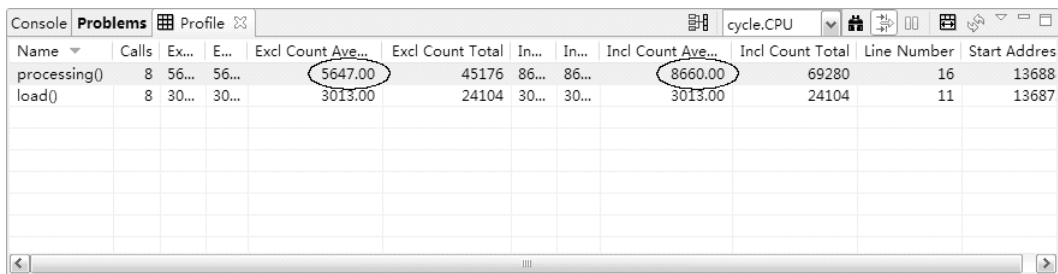
**例 5.4** 在前面的小节中，使用探针来模拟数据输入，使用图形显示工具观察运行结果。现在使用分析工具，统计这个程序中的 `processing()` 函数的执行情况。

(1) 装入编译、连接好的 OUT 文件，将程序执行到 `main()` 处。

(2) 在 Tools→Profile 菜单中，选“Setup Profile Data Collection”，新建立一个分析任务，其任务名可以由用户指定。勾选 Profile all functions for CPU cycles，单击 save 保存。

(3) 在项目管理窗口中双击 `main.c` 文件，在打开的源程序窗口中找到 `processing()` 函数，右击该行前面的状态列选择 New Breakpoint→Profile Control Point→Resume Profiling。然后，在其下面一行代码中同样添加一个 Profile Control Point，选 Pause Profiling。剖析点也是一个特殊的断点。

(4) 启动程序运行，可以单步、断点或连续运行。分析窗口的统计数据，仅在程序被 Halt 时才刷新。通过菜单 Tools→Profile→View Function Profile Results 可以打开统计结果窗口，如图 5.32 所示。统计结果主要有三个部分：第一部分是分析区间内，不包括子程序调用的执行时间 (Excl)；第二部分是整个分析区间 (Incl)，包括子程序调用的执行情况，如进入次数 (calls)、累计执行时间、最大、最小、平均执行时间；第三部分是这段分析代码的所在位置 (Line Number)、起始地址 (start address) 等信息。可以通过右击并选择 Column Settings 来设置显示的条目。例如：在 `processing()` 函数中调用了一个 `load()` 函数，其整个执行时间 (平均) 为 8 660 个机器周期，不包括子程序调用的执行时间 (平均) 为 5 647 个机器周期。



Name	Calls	Ex...	E...	Excl Count Ave...	Excl Count Total	In...	In...	Incl Count Ave...	Incl Count Total	Line Number	Start Address
processing()	8	56...	56...	5647.00	45176	86...	86...	8660.00	69280	16	13688
load()	8	30...	30...	3013.00	24104	30...	30...	3013.00	24104	11	13687

图 5.32 利用代码分析工具，分析代码执行时间

有关代码剖析工具中的其他分析方法，请参考 CCSv5 的 HELP 文档。

## 5.3 CCS 编程支持工具

CCS 编程支持工具包括：CMD 内存定位文件、DSP 片级支持库、DSP/BIOS 工具和 XDC 工具。本节将对其使用进行介绍。

### 5.3.1 CMD 内存定位文件的使用

TI 的 DSP 应用程序，是一段在固定地址运行的代码。也就是说，在编译、连接完成后，其代码的运行地址也就固定下来了。这与 PC 上的应用程序 (.exe 格式) 有很大差别。.exe 文件在装入时，由操作系统为其分配确定的内存地址 (可以从其跳转指令都采用相对地址跳转看出)。DSP 的应用程序不一样，当上电用 BOOTLOADER 加载，或用 CCS 装入命令



(Load program. . .) 装入时, 都会写入到指定的存储器中。这个指定的值, 是由用户在使用连接命令时提供的一个内存定位文件 (CMD 文件) 提供的。该 CMD 文件提供两个信息, 一个是系统有哪些内存块可供使用, 另一个是程序代码和数据将被安排到哪个内存块中。

### 1. 段的定义

为了有效地利用 DSP 的内部或外部存储器, 目标文件中的代码和数据分段存放。它们在内存中可以是分开的, 也可以是连续的。在生成的 out 文件中, 包括三个默认段 (已经定义了段名):

- .text 可执行代码;
- .data 已经初始化了的数据;
- .bss 为未初始化变量保存空间。

另外, 汇编器和连接器允许创建、命名和连接已经命名的段, 这些段可以像 .data、.text、.bss 段一样使用。

段有两种基本的类型。

- 已初始化的段: 即段中已经包括数据和代码。例如, .text 和 .data 段都是已初始化的段。在汇编语言中, 用户定义的、用于存放代码的 .sect 段, 也是已初始化的段。
- 未初始化的段: 是为未初始化数据保留的内存空间。例如, .bss 段是未初始化的, 用于为变量保留空间。在汇编语言中, 还可以使用 .usect 指令, 说明一个未初始化段。

汇编器在汇编过程中, 根据汇编伪指令创立这些段, 连接器确定这些段存放在目标存储器中的位置, 称为重定位。每个段都可独立重定位, 可以放在目标存储映射中任何已分配的段里。

下面以 C5000 的汇编语言为例, 简单说明如何使用段指令。其他系列的 DSP, 如 C2000 和 C6000, 使用方法是一致的。

**例 5.5:** 下面是一段有关段的使用的汇编语句, 其中文件列表中的每一行有四个域:

- Field 1 源代码行计数器;
- Field 2 段程序计数器;
- Field 3 目标代码;
- Field 4 最初的源语句。

```

2          ****
3          ** Assemble an initialized table into .data. **
4          ****
5 0000          .data
6 0000 0011    coeff          .word 011h,022h,033h
   0001 0022
   0002 0033
7          ****
8          ** Reserve space in .bss for a variable. **
9          ****
10 0000          .bss          buffer,10
11          ****

```

```

12          ** Still in .data. **
13          ****
14 0003 0123 ptr          . word      0123h
15          ****
16          ** Assemble code into the .text section. **
17          ****
18 0000          . text
19 0000 100f add:    LD      0Fh,A
20 0001 f010 aloop:  SUB     #1,A
    0002 0001
21 0003 f842          BC      aloop,AGEQ
    0004 0001V
22          ****
23          ** Another initialized table into .data. **
24          ****
25 0004          . data
26 0004 00aa ivals     . word      0AAh,0BBh,0CCh
    0005 00bb
    0006 00cc
27          ****
28          ** Define another section for more variables. **
29          ****
30 0000 var2     . usect    "newvars",1
31 0001 inbuf     . usect    "newvars",7
32          ****
33          ** Assemble more code into .text. **
34          ****
35 0005          . text
36 0005 110a mpy:    LD      0Ah,B
37 0006 f166 mloop:  MPY     #0Ah,B
    0007 000a
38 0008 f868          BC      mloop,BNOV
    0009 0006
39          ****
40          ** Define a named section for int. vectors. **
41          ****
42 0000          . sect     "vectors"
43 0000 0011          . word    011h,033h
1. 0001 0033

```

如图 5.33 所示，例 5.5 文件产生五个段。

.text 包含 10 个目标代码 Byte，为默认的程序代码段。

.data 包含 7 个目标代码 Byte，为默认的数据常量存放段。

**vectors** 包含 2 个 Byte 初始化数据, 是由 `.sect` 指令产生的、用户自己定义的程序代码段。如果需要, 用户可以自己定义多个这种代码段。

**.bss** 保留了 10 个 Byte 的存储器, 为默认的数据变量存放段。

**newvars** 保留了 10 个 Byte 的存储器, 是由 `.usect` 指令产生的、用户自己定义的数据段。如果需要, 用户可以定义多个这种数据段。

Line Numbers	Object Code	Section
19	100f	.text
20	f010	
20	0001	
21	f842	
21	0001	
36	110a	
37	f166	
37	000a	
38	f868	
38	0006	
6	0011	.data
6	0022	
6	0033	
14	0123	
26	00aa	
26	00bb	
26	00cc	
43	0011	vectors
44	0033	
10	No data— 10 words reserved	.bss
30	No data— eight words reserved	newvars
31		

图 5.33 段的划分

需要说明的是, 每个定义的段 (包括默认的和用户自己定义的) 都必须在 CMD 文件中为其指定确定的存储块。

## 2. 连接器如何使用段

连接器有两个与段有关的功能, 一个是在建立输出 out 文件时, 连接器要使用 obj 目标文件中的段, 合并输入段 (当连接的文件多于一个时); 另一个是连接器为输出段选择存储器地址。

有两条连接指令支持这些功能。

- 存储器指令 (memory directive): 定义目标系统的存储器映射。用户可以给部分存储器命名, 定义其起始地址和长度。
- 段指令 (sections directive): 告诉连接器如何将输入段合并到输出段中, 以及将这些输出段放在存储器的什么地方。用户可以用连接器的 sections 指令指定子段, 如不特别指定, 子段将和相同基段名的段合并在一起。

这些连接器指令并不一定要使用, 若不使用, 连接器就使用目标处理器的默认分配算法; 若要使用, 则必须在连接器的内存定位命令文件 (CMD 文件) 中说明。一般情况下, 都会为连接器提供一个这样的 CMD 文件。下面是一个常用的定位控制文件。

```

MEMORY
{
    PAGE 0: IPROG:  origin = 0x1800, len = 0x1f80
                        VECT:      origin = 0x200,      len = 0x100
    PAGE 1: IDATA1:    origin = 0xc00,      len = 0x100
                        IDATA2:    origin = 0xd00,      len = 0x100
                        IDATA3:    origin = 0xf00,      len = 0x100
}

SECTIONS
{
    .text:    {} > IPROG PAGE 0
    .vect     {} > VECT PAGE 0

    .bss      {} > IDATA1 PAGE 1
    .data     {} > IDATA2 PAGE 1
    .buffer   {} > IDATA3 PAGE 1
}

```

其中包含两个部分，MEMORY 和 SECTIONS。在 MEMORY 中，主要说明目标系统中哪些存储器可以使用，它们的起始地址和大小。例如，在上面的定义中，有两块程序存储空间，分别从 1800h 和 200h 开始，大小分别为 800h 和 100h；有三块数据储存空间，起始地址分别为 0c00h、0d00h、0f00h，大小分别为 100h、200h、100h。在这个 cmd 文件的第二个部分 SECTIONS 中，完成段的具体地址的分配。这个部分将给出程序中定义的所有段的具体安排，当然可以有程序中没有出现的段。例如，将 .text 段存放在 1800h 开始的程序空间中，将 .bss 段存放在 c00h 开始的数据空间中，将 .vect 段存放在程序空间的 200h 开始的地方。

上面仅对段的控制进行简要说明。下面来看一个完整的 C54x 汇编程序的编译、连接过程。

**例 5.6** 在 TMS320C542 上实现定时器中断，并通过 XF 引脚输出周期为 20Hz 的方波。为了实现该功能，使用两个汇编语言文件：asmlnk.asm 和 asmlnk1.asm。下面是程序源代码 asmlnk.asm 和 asmlnk1.asm：

```

-----
    . title " test asm - lnk. . ."
    . global _c_int00, _do_process
    . mmregs

; -----
;  define  data buffer in . bss section ( DATA MEMORY )
; -----

    . bss  wave_buf, 100h

; -----
;  define interrupt vectors table in . vectors section ( PROG MEMORY )
; -----

    . sect ". vect"

```

```

rs      b __c_int00
        nop
        nop
nmi     b __ret
        nop
        nop
sint17  b __ret
        nop
        nop
sint18  b __ret
        nop
        nop
sint19  b __ret
        nop
        nop
sint20  b __ret
        . word    0,0
sint21  b __ret
        . word    0,0
sint22  . word    01000h
        . word    0,0,0
sint23  . word    0ff80h
        . word    0,0,0
sint24  . word    01000h
        . word    0,0,0
sint25  . word    0ff80h
        . word    0,0,0
sint26  . word    01000h
        . word    0,0,0
sint27  . word    0ff80h
        . word    0,0,0
sint28  . word    01000h
        . word    0,0,0
sint29  . word    0ff80h
        . word    0,0,0
sint30  . word    01000h
        . word    0,0,0
int0    b __ret
        nop
        nop
int1    b __ret
        nop
        nop

```

```

int2      b __ret
          nop
          nop
tint      b timer
          nop
          nop
brint0    b __ret
          nop
          nop
bxint0    b __ret
          nop
          nop
trint     b __ret
          nop
          nop
txint     b __ret
          nop
          nop
int3      b __ret
          nop
          nop
hpint     b __ret
          nop
          nop
q26       . word    0ff80h
          . word    0,0,0
q27       . word    01000h
          . word    0,0,0
q28       . word    0ff80h
          . word    0,0,0
q29       . word    01000h
          . word    0,0,0
q30       . word    0ff80h
          . word    0,0,0
q31       . word    01000h
          . word    0,0,0

          . text
_c_int00:
          ssbx intm          ; close all int ! (ssbx intm)
          stm  #10h,26h      ; stop TIMER0 !
          stm  #10h,36h      ; stop TIMER1 !
          stm  #0ffh,sp      ; sp = 0x0ff

```



```

        ld    #0,dp          ; dp=0
        stm   #0820h,pmst    ; vector table start: 0x800
; *****
; initialize RAM,Variable...
; *****
        stm   #wave_buf,ar6
        rpt   #0fffh
        st    #0,*ar6 +      ; clear data buffer TO 0 !
; *****
; The following codes are used to initialize TIMER !
; *****
        stm   #10h,tcr        ; stop TIMER !
        stm   #0fffh,prd      ;
        stm   #0fh,tcr        ; TIMER start,each about 40HZ...
        stm   #8,imr          ; enable TIMER INT !
again:
        bitf  st1,#2000;      test is_new_data == 1 ?
        cc    _do_process,tc   ; ==1,then call do_process !
        b     again
; *****
; The following codes are served for vc5402V TIMER !
; *****
timer:
        pshm  ah
        pshm  al

        bitf  st1,#2000h      ; test XF?
        bc    show_led,tc
        ssbx  xf              ; set xf = 1
        b     show_con
show_led:
        rsbx  xf              ; set xf = 0
show_con:
        popm  al
        popm  ah
__ret:
        rete
        end

```

asmlnk1.asm 源代码。

```

        global _do_process
; *****

```

```

; The following codes is only to simulate function call !
; *****
_do_process:
    pshm ar3
    rpt #0ffh
    nop
    popm ar3
    ret
-----

```

asmlnk.asm 包含了两个程序代码段，.text 和 .vect。其中，.vect 段用来存放中断向量表，.text 段存放初始化代码、中断服务程序等。另外，还在数据空间的 .bss 段中，定义了一个数据缓冲 wave\_buf，长度为 100h。asmlnk1.asm 中存放了一些处理代码，每次 XF 为 1 时运行。由于篇幅有限，这里仅保留了处理程序的框架。该代码也存放在程序空间的 .text 段中。所有这些段的具体地址分配，应该在连接器工作时，通过内存映射 cmd 文件确定。首先使用汇编器完成汇编，使用 -s 选项（保存所有符号），分别生成 asmlnk.obj 和 asmlnk1.obj。

```

asm500 asmlnk1.asm asmlnk1.obj -s

TMS320C54x COFF Assembler          Version 3.50
Copyright (c) 1996 - 1999 Texas Instruments Incorporated

PASS 1
PASS 2
No Errors, No Warnings

asm500 asmlnk.asm asmlnk.obj -s

TMS320C54x COFF Assembler          Version 3.50
Copyright (c) 1996 - 1999 Texas Instruments Incorporated

PASS 1
PASS 2
No Errors, No Warnings

```

在成功完成汇编工作后，便可以使用连接器来连接。在使用连接器之前，应该根据具体的程序运行环境，编写或修改内存映射 cmd 文件。这个程序将在 VC542 的片内 RAM 运行。VC542 片内有 10K 字的 RAM，地址为 0 ~ 27FFH。由于片内 RAM 的程序空间和数据空间共享这 10K 字，且 0 ~ 5FH 被系统内存映射存储器占用，所以将这 10K 空间分为程序和数据两个部分，60H ~ 7FFH 为数据空间，800H ~ 27FFH 为程序空间。下面是 demo542.cmd 文件。

```

asmlnk.obj
asmlnk1.obj
-m asmlnk.map
-o asmlnk.out
MEMORY
{

```

```

PAGE 0: IPROG:      origin = 0x880,      len = 0x1f80
          VECT:      origin = 0x800,      len = 0x80
PAGE 1: USERREGS:   origin = 0x60,      len = 0x1c
          BIOSREGS:   origin = 0x7c,      len = 0x4
          IDATA:      origin = 0x80,      len = 0x780
}

SECTIONS
{
    . vect:          {} > VECT PAGE 0
    . sysregs: {} > BIOSREGS PAGE 1
    . trcinit:       {} > IPROG PAGE 0
    . gblinit:       {} > IPROG PAGE 0
    . biOS: {} > IPROG PAGE 0
    . frt:           {} > IPROG PAGE 0
    . text:          {} > IPROG PAGE 0
    . cinit:         {} > IPROG PAGE 0
    . pinit:         {} > IPROG PAGE 0
    . sysinit:       {} > IPROG PAGE 0
    . bss: {} > IDATA PAGE 1
    . far:           {} > IDATA PAGE 1
    . const:         {} > IDATA PAGE 1
    . switch:        {} > IDATA PAGE 1
    . system: {} > IDATA PAGE 1
    . cio:           {} > IDATA PAGE 1
    . MEM $obj: {} > IDATA PAGE 1
    . sysheap: {} > IDATA PAGE 1
}

```

在程序空间（PAGE 0）中，定义了两个部分，一个专门用于存放中断向量表。在连接完成后，中断向量表将放在 800H 开始的地方。其他代码段如 .text、.cinit 等，都依次安排到 880H 开始的存储器中。在这个例子中，仅仅使用了 .text 段。数据空间分为三个部分，其中 BIOSREGS 保留来存放 CCS 中 DSP/BIOS 的变量。其他定义的数据段，如 .bss、.const 等，依次安排在从 80H 开始的空间中。同样，这个例子也仅仅使用了 .bss 段。另外，内存映射 demo542.cmd 文件中，不仅可以定义代码、数据地址的分配，还可以将连接器使用的参数放入（若使用 CCS 开发环境，这些参数将被保存到 mak 文件中），例如需要生成 map 文件。调用连接器可以简单地使用如下命令。

```
lnk500 demo542.cmd
```

```

TMS320C54x COFF Linker          Version 3.50
Copyright (c) 1996 - 1999 Texas Instruments Incorporated

0 Errors,0 Warnings.

```

下面是连接器生成的 map 文件，通过它可以了解程序中各个段、子程序、变量、程序入口等地址信息，从而直观地了解程序的存储器占用情况。例如，程序空间总共占用 30h，代码入口地址 880h，子程序\_do\_process 入口地址 8abh，等等。这时，可以用调试工具将生成的 asmlnk.out 装入 VC542 的片内 RAM，开始调试工作。

```

TMS320C54x COFF Linker          Version 3.50
>> Linked Sun Oct 07 10:31:40 2001
OUTPUT FILE NAME:  < asmlnk.out >
ENTRY POINT SYMBOL: "_c_int00"  address: 00000880

MEMORY CONFIGURATION

      Name      origin      length      used      attributes      fill
      -----      -
PAGE 0: VECT      00000800      00000080      00000080      RWIX
      IPROG      00000880      000001f8      00000030      RWIX

PAGE 1: USERREGS 00000060      0000001c      00000000      RWIX
      BIOSREGS      0000007c      00000004      00000000      RWIX
      IDATA          00000080      00000078      00000100      RWIX

SECTION ALLOCATION MAP

Output
Section      page      origin      length      attributes/
      -----      -
. vect      0      00000800      00000080
      00000800      00000080      asmlnk. obj (. vect)

. sysregs  1      0000007c      00000000      UNINITIALIZED
. trcinit   0      00000880      00000000      UNINITIALIZED
. gblinit   0      00000880      00000000      UNINITIALIZED
. biOS      0      00000880      00000000      UNINITIALIZED
Frt         0      00000880      00000000      UNINITIALIZED
. text      0      00000880      00000003
      00000880      0000002b      asmlnk. obj (. text)
      000008ab      00000005      asmlnk1. obj (. text)
. cinit     0      00000880      00000000
. pinit     0      00000880      00000000
. sysinit   0      00000880      00000000      UNINITIALIZED
. bss       1      00000080      00000100      UNINITIALIZED
      00000080      00000100      asmlnk. obj (. bss)
      00000180      00000000      asmlnk1. obj (. bss)
. far       1      00000080      00000000      UNINITIALIZED
. const     1      00000080      00000000      UNINITIALIZED

```

```

. switch      1      00000080      00000000      UNINITIALIZED
. sysmem 1    00000080      00000000      UNINITIALIZED
. cio      1    00000080      00000000      UNINITIALIZED
. MEM $obj    1      00000080      00000000      UNINITIALIZED
. sysheap 1    00000080      00000000      UNINITIALIZED
. data      1    00000000      00000000      UNINITIALIZED
                                00000000      00000000      asmlnk. obj (. data)
                                00000000      00000000      asmlnk1. obj (. data)

```

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address      name

(以下省略)

[ 19 symbols ]

### 5.3.2 DSP 片级支持库

在 TI 公司的 CCS 开发环境中, 提供了 DSP 片级支持库 CSL, 即 Chip Support Library, 提供一系列应用程序 C 调用接口 (API), 用于配置和控制 DSP 片上外设。多数 CSL 模块都由对应函数、宏、类和表示符号组合构成, 在不调用其他 DSP/BIOS 组件的情况下, 也可以方便地完成对 DSP 器件片上外设的配置和控制, 从而简化开发工作。它还使片上外设的管理标准化, 减少不同 DSP 硬件对用户程序的影响, 以方便用户代码在不同器件间的移植。TI 为每个型号的 DSP 都提供了 CSL 库, 用户可以在 TI 的网站上免费下载。下面给出的是 C5400 系列提供的 CSL 模块库, 见表 5.1。对于其他系列, 如 C5500、C6000 等, 都有相应的 CSL 库。表中, CSL GUI 支持一项, 表示该模块是否可以使用 CSL 图形化用户接口 GUI 来控制。所列出的头文件, 要在使用这些模块的应用程序中加以引用。

表 5.1 C5400 CSL 模块和对应的头文件

外设模块	功能描述	头文件名	模块支持符号	CSL GUI 支持
CHIP	通用器件模块	Csl_chip. h	_CHIP_SUPPORT	不
DAA	数据访问分派模块	Csl_daa. h	_DAA_SUPPORT	不
DAT	基于 DMA 的数据复制、填充模块	Csl_dat. h	_DAT_SUPPORT	不
DMA	DMA 外设模块	Csl_dma. h	_DMA_SUPPORT	支持
EBUS	外部总线接口模块	Csl_ebus. h	_EBUS_SUPPORT	支持
GPIO	GPIO 模块	Csl_gpio. h	_GPIO_SUPPORT	支持
HPI	HPI 外设模块	Csl_hpi. h	_HPI_SUPPORT	不
IRQ	中断控制模块	Csl_irq. h	_IRQ_SUPPORT	不
MCBSP	MCBSP 模块	Csl_mcbbsp. h	_MCBSP_SUPPORT	支持
PLL	PLL 模块	Csl_pll. h	_PLL_SUPPORT	支持

续表

外设模块	功能描述	头文件名	模块支持符号	CSL GUI 支持
PWR	节能控制模块	Csl_pwr. h	_PWR_SUPPORT	不
TIMER	TIMER 模块	Csl_timer. h	_TIMER_SUPPORT	支持
UART	UART 串口模块	Csl_uart. h	_UART_SUPPORT	支持
WDTIM	看门狗定时器模块	Csl_wdtim. h	_WDT_SUPPORT	支持

不同种类的 DSP 器件，需要不同的 CSL 库来支持。表 5.2 描述了 CSL 库对 C54x 系列 DSP 器件的支持情况。使用 DSP 器件支持符号的时候，编译器要加入编译参数 -d，以保证这些支持符号的正确关联。

表 5.2 支持 C54x 的 CSL

DSP 器件型号	近调用模式库	远调用模式库	DSP 器件支持符号表示
C5401	Csl5401. lib	Csl5401x. lib	CHIP_5401
C5402	Csl5402. lib	Csl5402x. lib	CHIP_5402
C5404	Csl5404. lib	Csl5404x. lib	CHIP_5404
C5407	Csl5407. lib	Csl5407x. lib	CHIP_5407
C5409	Csl5409. lib	Csl5409x. lib	CHIP_5409
C5409A	Csl5409A. lib	Csl5409Ax. Lib	CHIP_5409A
C5410	Csl5410. lib	Csl5410x. lib	CHIP_5410
C5410A	Csl5410A. lib	Csl5410Ax. lib	CHIP_5410A
C5416	Csl5416. lib	Csl5416x. lib	CHIP_5416
C5420	Csl5420. lib	Csl5420. lib	CHIP_5420
C5421	Csl5421. lib	Csl5421x. lib	CHIP_5420
C5440	Csl5440. lib	Csl5440x. lib	CHIP_5440
C5441	Csl5441. lib	Csl5441x. lib	CHIP_5441
C5471	Csl5471. lib	Csl5471x. lib	CHIP_5471
C54cst	Csl54cst. lib	Csl54cstx. lib	CHIP_54CST

**例 5.7** 使用 CSL 库，在 C5402 上完成一个 DMA 通道的配置。本例要实现的功能是：初始化 DMA channel0，使用寄存器配置方式，完成从地址 0X900 到地址 0X910 的数据表格的复制。本例程演示的是在 CCSv3.3 下的实现过程，CCSv5 有相似设置。

源地址：数据空间 900H

目的地址：数据空间 910H

传输数据量：16 个 16-bit 字

**第一步：**开启 CCS 环境，建立项目 dma1a，添加应用程序代码 dma1a.c。

```

/* 包含所需 CSL 头文件 */
#include <csl. h>
#include <csl_dma. h>
/* -----*/
/* 初始化例程参数 */

```



```

#define N      16                                //指定传输块大小
#pragma DATA_SECTION( src, " dmaMem" )          //指定源数据表地址
Uint16 src[ N ] = {
    0xBEEFu, 0xBEEFu, 0xBEEFu, 0xBEEFu, 0xBEEFu, 0xBEEFu,
    0xBEEFu, 0xBEEFu, 0xBEEFu, 0xBEEFu, 0xBEEFu, 0xBEEFu,
    0xBEEFu, 0xBEEFu, 0xBEEFu, 0xBEEFu
};                                                //建立源数据表
#pragma DATA_SECTION( dst, " dmaMem" )          //指定目的数据表地址
Uint16 dst[ N ];                                //建立目的数据表
/* 使用内部存储器的数据空间到数据空间 DMA 传输模式 */
/* 设定 DMA 模式控制寄存器 DMMCR */
/*   DMMCR0 = 0x0145u                                */
/*   #00000000101000101b                                */
/*   ;0 ~~~~~~ ( AUTOINIT)   No Autoinit                */
/*   ; ~0 ~~~~~~ ( DINM)      No Interrupts              */
/*   ; ~0 ~~~~~~ ( IMOD)      N/A                        */
/*   ; ~0 ~~~~~~ ( CTMOD)     Multi - frame on           */
/*   ; ~0 ~~~~~~                N/A                      */
/*   ; ~001 ~~~~~~ ( SIND)     Src addr Post - incr      */
/*   ; ~01 ~~~~~~ ( DMS)       Src in data space          */
/*   ; ~0 ~~~~~~                N/A                      */
/*   ; ~001 ~~~ ( DIND)       Dst addr Post - incr       */
/*   ; ~01 ~~~ ( DMD)         Dst in data space           */
/* 设定 DMA 同步和帧计数寄存器 DMSFC                                */
/*   DMSFC0 = 0x0000u                                */
/* */
/*   #0000000000000000b                                */
/*   ;0000 ~~~~~~ ( DSYN)     No sync event              */
/*   ; ~0 ~~~~~~ ( DBLW)      Single - word mode         */
/*   ; ~000 ~~~~~~                N/A                    */
/*   ; ~00000000 ( Frame Count) = 0 ( one frame)        */
/* 使用 CSL 宏和符号常量完成对上述两个寄存器的预定义, 创建 DMA 通道配置结构 */
DMA_Config myconfig = {
    1,                                                /* 设置优先级 */
    DMA_DMMCR_RMK(
        DMA_DMMCR_AUTOINIT_OFF,
        DMA_DMMCR_DINM_OFF,
        DMA_DMMCR_IMOD_FULL_ONLY,
        DMA_DMMCR_CTMOD_MULTIFRAME,
        DMA_DMMCR_SIND_POSTINC,
        DMA_DMMCR_DMS_DATA,
        DMA_DMMCR_DIND_POSTINC,
        DMA_DMMCR_DMD_DATA
    )
};

```

```

    ),                                     /* DMMCR */
    DMA_DMSFC_RMK(
        DMA_DMSFC_DSYN_NONE,
        DMA_DMSFC_DBLW_OFF,
        DMA_DMSFC_FRAMECNT_OF(0)
    ),                                     /* DMSFC */
    (DMA_AdrPtr) &src[0],                 /* DMSRC */
    (DMA_AdrPtr) &dst[0],                 /* DMDST */
    (Uint16)(N - 1)                       /* DMCTR */
};
/* TASK 函数声明 */
void taskFunc( void );
void main( ) {
    Uint16 i;
    /* 必须完成对 CSL 库的初始化工作以后才能调用 CSL 模块的 API */
    CSL_init();
    /* 调用 TASK 函数 */
    taskFunc();
}
/* -----*/
void taskFunc( void ) {
    Uint16 err = 0;
    Uint16 i;
    /* 定义 DMA_Handle 指针,DMA_open 将对其进行初始化操作 */
    DMA_Handle myhDma;
    /* 打开 DMA channel 0 来完成传输工作 */
    /* 在多资源外设中如 McBSP,DMA 将使用 PER_open 函数来打开资源 */
    myhDma = DMA_open( DMA_CHA0,0 );
    /* 调用 DMA_config() 函数配置 DMA */
    DMA_config( myhDma,&myconfig );
    /* 调用 DMA_start() 函数 启动传输 */
    DMA_start( myhDma );
    while( DMA_getStatus( myhDma ) );
    /* 确认传输数据是否正确 */
    for ( i = 0; i <= N - 1; i ++ ) {
        if ( dst[i] != 0xBEEFu ) {
            ++ err;
        }
    }
    /* 关闭 DMA 通道 */
    DMA_close( myhDma );
}

```

**第二步：**在 CCS 开发环境中指定目标设备。选择 Project→Build Options，打开 Build Options对话框，选取 Compiler 编译器，参照图 5.34 设定器件类型。例如，C5402 DSP 芯片，需键入 CHIP\_5402，并确定。



图 5.34 在 CCS 中定义目标器件

**第三步：**根据使用的 CSL 函数库的类型（远模式库/近模式库），选择相关选项，并确定存储器大小模式。如果使用了大存储器模式的函数库，必须在编译器中选择 -ml 参数，并且在连接器中键入相关路径，如图 5.35 所示。

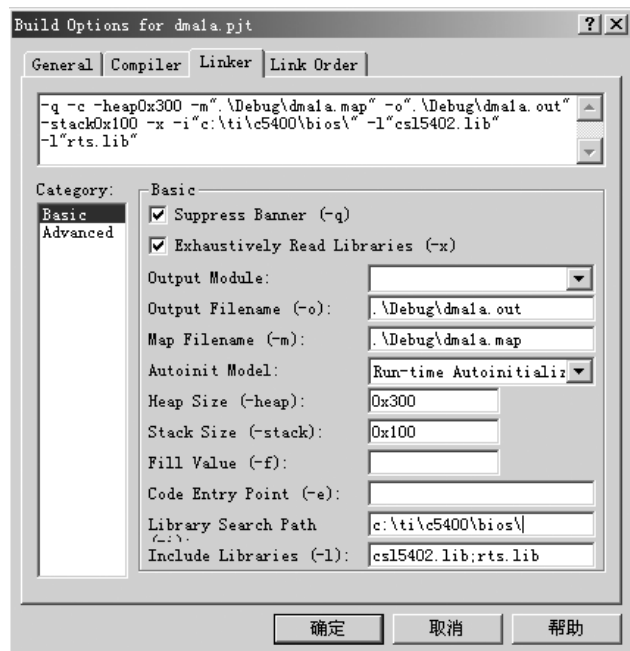


图 5.35 定义函数库路径

**第四步：**创建连接命令字文件（.cmd 文件）。CSL 要求将其特殊数据段 .csldata 配置在 64K 的基本数据空间中，并且保留数据空间中写字板存储器的 0X7B 地址单元，它在执行 CSL\_init() 进行 CSL 初始化操作的时候作为指向 .csldata 段的指针。基于这一原因，在用户程序中调用 CSL\_init() 函数，必须先于调用其他 CSL 函数，否则可能因为写入 0X7B 地址单元而使调用 CSL 函数失败。下面是 dma1.cmd 文件，阅读时注意上面讨论的两种情况。

```
MEMORY {
    PAGE 0:  VECT:      origin = 0xff80,      len = 0x80
    PAGE 1:  IDATA:     origin = 0x80,        len = 0x880
    PAGE 0:  IPROG:     origin = 0xA00,       len = 0x2800
    PAGE 1:  EDATA:     origin = 0x8000,      len = 0x8000
    PAGE 0:  EPROG:     origin = 0x8000,      len = 0x7f80
    PAGE 1:  DMAMEM:    origin = 0x900,       len = 0x100
} /* MEMORY 没有分配 0X7B 数据单元 */

SECTIONS {
    .text      > IPROG PAGE 0          /* code          */
    .switch    > IPROG PAGE 0          /* switch table info */
    .cinit     > IPROG PAGE 0
    .vectors   > VECT PAGE 0           /* interrupt vectors */

    .cio       > IDATA PAGE 1          /* C I/O          */
    .data      > IDATA | EDATA PAGE 1  /* initialized data */
    .bss       > IDATA | EDATA PAGE 1  /* global & static variables */
    .const     > IDATA PAGE 1          /* constant data   */
    .sysmem    > IDATA PAGE 1          /* heap            */
    .stack     > IDATA PAGE 1          /* stack           */
    .csldata   > IDATA PAGE 1

    DmaMem: align(256) { } > DMAMEM PAGE 1
} /* SECTIONS 为 .csldata 段给出了定位信息 */
```

**第五步：**完成编译、连接操作，并运行。选择 VIEW→MEMORY 对话框，坚持源数据地址内容与目标数据地址内容是否一致。

片级支持库 CSL，可以近似地看作 DSP 器件的片上外设驱动，因而和 DSP 器件型号密切相关。不同芯片的 CSL 不同，CSL 包括的各个 CSL 模块也不相同。读者可以参考有关文献和资料。

### 5.3.3 DSP/BIOS 工具的使用

3.1 节已经讨论了 DSP/BIOS 实时内核的概念和组件。DSP/BIOS 为开发者提供了一系列运行时服务的内核。DSP/BIOS 内核有效地扩展了 DSP 指令集，包括实时服务和运行时内核服务，构成了实时 DSP 应用的底层体系架构和基础架构。在 DSP 程序的开发过程中，DSP/BIOS 工具不是必备的，完全可以使用前面介绍的方法，在 CCS 常规环境下，开发自己的

DSP 应用程序。但是, DSP/BIOS 工具可以帮助软件工程师更加容易地控制 DSP 的硬件资源,更加灵活地协调各个软件模块的执行,更加清楚地了解程序的运行状况,包括 DSP 的动态负荷、各个线程之间的关系等,大大加快软件的开发和调试进度。本节将简单介绍如何利用与 CCS 集成开发环境一起发布的 DSP/BIOS 工具,完成基于 DSP/BIOS 的应用程序的调试。

### 5.3.3.1 建立 DSP/BIOS 的配置文件

每个使用 DSP/BIOS 的程序,都需要一个 DSP/BIOS 的配置文件,并将该配置文件添加到项目文件中。根据该配置文件,系统自动生成 LNK 使用的 .cmd 文件和相关的汇编代码(这些汇编或 cmd 文件也必须添加到项目文件中)。可以通过 DSP/BIOS 配置工具,新建或修改一个配置文件。该配置文件可以定义程序运行中的初始化系统参数,如 PMST、SWWSR、CLKMD 等寄存器。另外,在配置文件中,还可以使用可视化的工具,定义或编辑程序需要的各种 DSP/BIOS 提供的模块对象,如中断、流水、事件记录、时钟、周期函数等,见图 5.36。同时, BIOS 的配置工具也提供了文本方式的配置工具 (\* .tcf)。

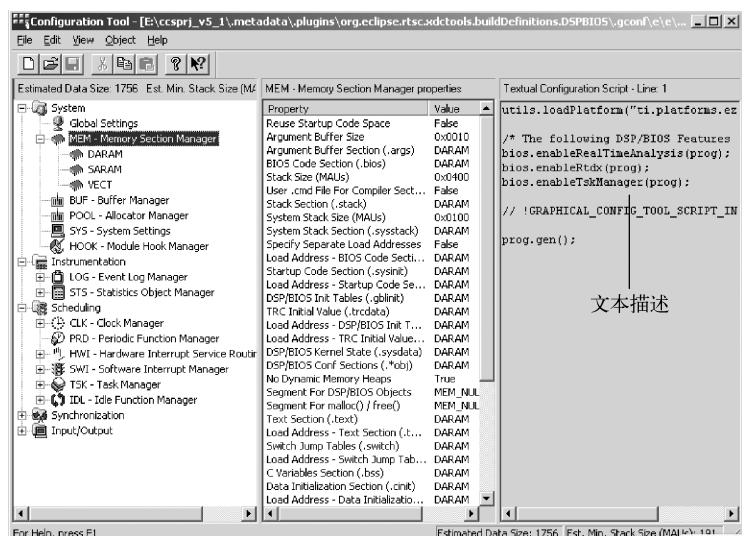


图 5.36 编辑 BIOS 的配置文件 (\* .tcf)

新建一个 DSP/BIOS 的配置文件,方法如下:在 CCS 的“File”菜单中,选择“New→DSP/BIOS Configuration”。现在的 BIOS 中提供了非常多的硬件平台模板,用户可以选择使用的 EVM 板或相近的模板来创建。配置文件中有一个 MEM 模块,当保存该 BIOS 配置文件时,会自动产生一个用于连接的内存定位 cmd 文件。该 cmd 文件根据配置文件的 MEM 段的内容来决定。单击“MEM - Memory Section Manager”,所有的内存段定义分配信息都显示在左边窗口中。要修改这些段的分配,可右击“MEM”,然后选“Properties”。在图 5.36 所示的 MEM 管理中,已经预定义几个内存块,其中 VECT 用于存放中断向量表,SDRAM 和 SARAM 分别表示片内的双寻址 RAM 和单寻址 RAM。不同的目标板,可以有不同的内存块,这与创建配置文件时选择的平台有关。当然也可以自己定义一块存储器。一般情况,如果你的硬件存储器配置情况与模板一致,就不用修改。否则需要根据自己的硬件环境做修改。当保存修改后的配置文件时,配置文件编辑工具,还将自动生成 xxxxcfg.h、xxxxcfg.cmd 等

DSP/BIOS 应用程序所需要的系统文件。其中“xxxx”代表配置文件的名字，这些 .h 文件定义了 BIOS 中定义的各个模块、对象，而 .cmd 文件是内存定位文件。

### 5.3.3.2 用 DSP/BIOS 工具创建应用程序

使用 DSP/BIOS 工具来开发 DSP 应用程序，与前面介绍的不使用 DSP/BIOS 工具来开发的步骤基本一致。首先，在 Project 菜单中选择“New CCS Project”或“Import...”，新建或打开一个项目文件。然后，将需要的 .h、.asm、.c、.obj、.lib（这时不需要添加 C 语言标准库，如 rts.lib）添加到该项目文件中。还需要将 DSP/BIOS 的配置文件 .cdb 或 .tcf 添加到项目文件中。注意：连接工具使用的 .cmd 文件，由 DSP/BIOS 配置文件自动产生，在 CCSv5 下连接工具会自动添加 DSP/BIOS 配置工具生成的 .cmd 文件。下面，通过实际的例子来说明 DSP/BIOS 工具的使用。

**例 5.8** 本例将介绍如何创建一个基于 DSP/BIOS 的应用程序，在 DSP/BIOS 中如何静态创建对象（object），如何创建静态 SWI 线程，以及 SWI 线程调用等。

在这个例子中，主要有下面两个源文件：volume\_bios.c 和 load.c。下面是两个文件的内容。

```

/* -----volume_bios.c ----- */
/* Global declarations */
Int inp_buffer[ BUFSIZE];      /* processing data buffers */
Int out_buffer[ BUFSIZE];
Int gain = MINGAIN;            /* volume control variable */
Uns processingLoad = BASELOAD; /* processing routine load value */
/* Functions */
extern Void load( Uns loadValue);
Int processing( Int *input, Int *output);
Void dataIO( Void);
/* ===== main ===== */
Void main()
{
    LOG_printf( &trace, "new volume example started\n" );
    /* fall into DSP/BIOS idle loop */
    return;
}
/* ===== processing ===== */
Int processing( Int *input, Int *output)
{
    Int size = BUFSIZE;
    while( size -- ) {
        *output ++ = *input ++ * gain;
    }
    /* additional processing load */
    load( processingLoad);
}

```



```

        return(TRUE);
    }
    /* ===== dataIO ===== */
    Void dataIO()
    {
        /* do data I/O */
        SWI_post(&processing_SWI); /* post processing_SWI software interrupt */
    }

    /* -----load.c ----- */
    int load(int processload)
    {
        int i,j,k;
        for(i=0;i<processload;i++)
        {
            for(j=0;j<100;j++)
                for(k=0;k<10;k++);
        }
        return(1);
    }

```

在上面的两个 C 文件中，共有四个函数：main()、processing()、dataIO()以及 load()。其中 processing()函数是主处理函数，完成将输入缓冲 input 的数据乘以一个增益 (gain)，再调用 load()函数。Load()函数是一个延时函数，目的是仿真算法处理需要一定的时间。而 dataIO()函数来模拟中断服务处理，在数据准备好后启动了一个 SWI 线程。

在下面的例子中，将 dataIO()放到 CLOCK 模块中，每隔 1ms 调用一次，来模拟外部中断。同时，将 processing()函数放到 processing\_SWI 软件中断线程中执行。这样，相当于每 1ms 执行一次 processing()处理函数。

- 在 CCSv5 新建一个项目文件，比如命名为 bios。在 New 菜单下选择 CCS Project。由于采用的硬件平台是 USBSTK5505，所以选择 5500 系列。如果使用 DES6437 的平台，请选择 C6000 以及 C64 + device。单击“Finish”后，CCS 会创建一个项目文件。
- 新建的项目文件只有一个空的 main.c，将其删除。可以将 load.c 和 volume\_bios.c 复制到新建的 bios 项目文件夹中，CCS 会自动将这个两 C 代码添加到项目中。这两步与普通 CCS 下开发代码没有区别。
- 创建 BIOS 的配置文件。每个使用 DSP/BIOS 的项目，都必须包含一个 BIOS 配置文件。右击 bios 项目文件，在弹出菜单中选择 New→Other.、选择展开 RTSC、选择“DSP/BIOS v5. x Configuration File”，然后单击“Next”，确认配置文件的文件名后，继续。为了方便说明，定义的配置文件名为 bios\_5505.tcf。再次单击“Next”后，如图 5.37 所示。这里需要选择 DSP/BIOS 配置文件所依赖的硬件平台，也就是将要运行 BIOS 代码的目标平台。由于使用 USBSTK5505 评估板，选择“ti.platforms.ezdsp5505”。如果使用 DES6437，则应该选择“ti.platforms.evmDM6437”。配置文件于是创建完成。

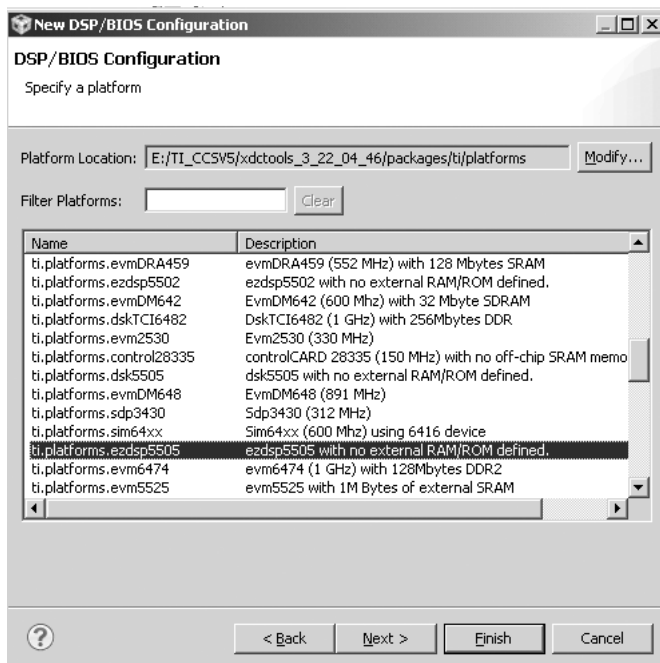


图 5.37 选择 DSP/BIOS 的配置平台

- DSP/BIOS 配置文件（这里创建的是 bios\_5505.tcf）将自动生成一系列的文件。其中，bios\_5505cfg.h（在 debug 目录下）定义了配置文件中创建的所有对象，以及包含了相关的 DSP/BIOS 头文件、API 函数说明、变量说明，使用时应该将其包含到 C 代码中。打开 volume\_bios.c，在前面添加如下代码，包含相关头文件：

```
#include "bios_5505cfg.h"
```

而其他生成的文件，CCS 都会自动引用。

- 在 DSP/BIOS 环境下，虽然可以使用标准的 C 函数，如标准输入/输出函数 printf、puts 等，但它们占用的空间较大，执行的效率也较低。所以，推荐使用 DSP/BIOS 提供的 API。在 volume\_bios.c 中，将使用 BIOS 提供的标准输出函数 LOG\_printf() 函数。
- 使用 DSP/BIOS 工具进行开发时，用户的应用程序必须建立在 DSP/BIOS 的控制下。所以，用户的应用程序应该尽快将 CPU 的控制权交回 DSP/BIOS 系统，即尽早从 main() 函数返回。如果用户的应用程序，或某个函数出现死循环而无法返回时，系统将瘫痪。所以，volume\_bios.c 的 main() 函数完成打印后，便结束返回。
- 使用 DSP/BIOS 提供的 LOG 模块，替代标准输入/输出函数。LOG 模块是 DSP/BIOS 提供的、专门用于记录输入/输出信息的模块。这个模块的调用，与 DSP/BIOS 提供的其他功能模块的调用有相似之处。与标准 C 的输出函数相比，使用 LOG 模块占用的内存更少，执行的速度更快。使用 DSP/BIOS 的 LOG 模块，依照下面步骤：
- 在项目文件窗口下双击配置文件，将其打开，展开“Instrumentation”项下的 LOG 选项（LOG 模块中已经有一个 LOG\_system，这是 DSP/BIOS 系统保留的，不要随意删除）并右击，在弹出菜单中选择“Insert LOG”项。右击新增加的 LOG0，修改新增 LOG0 对象的名称，如 trace。然后右击，并在弹出菜单中选择“Properties”，修改其参数，如将 buflen 改为 512。另外，将 LOG\_system 的 buflen 项也改为 512，见

图 5.38。

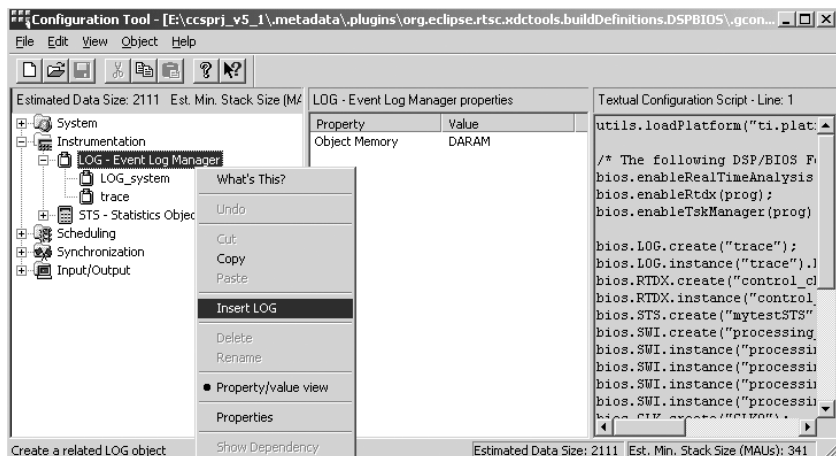


图 5.38 修改配置文件，增加 LOG 模块

- 这样在 C 源程序中的 main() 函数中，就可以用 LOG API 函数，如

```
LOG_printf(&trace, "new volume example started\n");
```

在 DSP/BIOS 提供的事件记录窗口显示信息（等效于标准 C 中的 printf 函数）。

- 在添加了 LOG 对象 trace 后，继续修改 BIOS 配置文件。展开 Scheduling，在 CLK 模块下添加一个对象 CLK0，并在 Properties 的菜单中修改 function 参数为\_dataIO。这表明，在每 1ms 一次的定时器时钟中断中，每 1ms 要运行 dataIO() 函数一次。
- 添加 SWI 线程。展开 Scheduling，在 SWI 模块下添加一个对象，并改名字为 processing\_SWI。这就是新创建的一个 SWI 线程。进入 Properties 属性菜单，设置线程调用函数为\_processing；在 arg0 和 arg1 两个参数位置，分别设置 processing() 函数调用时需要传递的两个参数：\_inp\_buffer 和 \_out\_buffer。这两个参数在 volume\_bios.c 中定义为两个全局数组，processing() 函数将 inp\_buffer[] 的数据乘以 gain 后放到 out\_buffer[] 中，如图 5.39 所示。

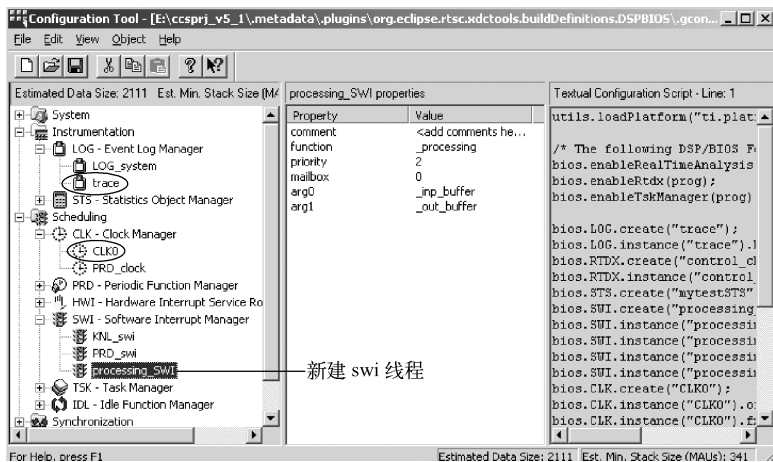


图 5.39 修改配置文件，增加 CLK 和 SWI 线程

- 到此，本例程的项目文件全部创建完成。可以右击项目名，在弹出菜单中选 Build Project 完成编译、连接。
- 打开 debug 窗口，装入项目生成的 bios.out（在 debug 目录下），就可以准备调试、运行了。为了看到 BIOS 的 LOG\_printf 打印输出，需要选择 Tools→RTA→Printf Logs，以便显示输出信息。另外，还可以选择 RTA→CPU Load 察看 CPU 的负载情况，如图 5.40 所示。

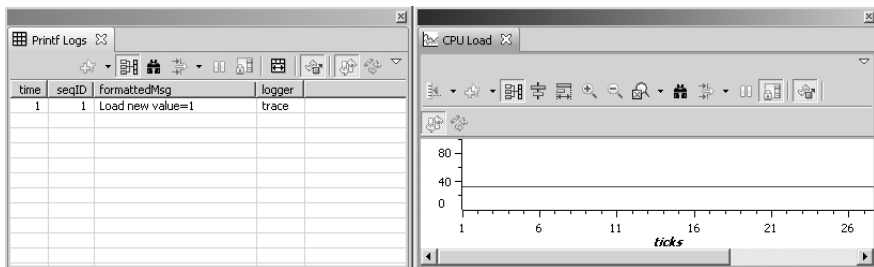


图 5.40 LOG 输出和 CPU 负载图

- 在 CCSv5 下，还有一些实时分析工具可以使用（RTA 下），比如 Statistics Data（统计数据显示）、Raw Logs（系统所有输出信息）等。但 CCSv5 下，去掉了 Execute Graph（线程状态执行图）功能。这个功能只有在 SYS/BIOS 提供。但 CCS3.3 的 BIOS 下仍然可以提供线程状态执行图。

### 5.3.3.3 DSP/BIOS 系统工具的使用

前面已经介绍了 DSP/BIOS 内核的基本情况和使用方法，包括配置文件的建立以及包含 DSP/BIOS 内核的 DSP 应用程序的基本开发过程。从例 5.8 可以看出，DSP/BIOS 内核的使用，简单地讲，就是 DSP/BIOS 提供的 API 模块的调用过程。在后面的章节中，会继续深入介绍 DSP/BIOS 的 API 调用。这里先介绍 CCS 为使用了 DSP/BIOS 内核的应用程序，提供了哪些系统分析工具。

#### 1. DSP/BIOS 工具控制面板

为了显示 RTA 控制面板，用户可以在 CCS 中的 Debug 窗口的菜单栏中选择 Tools→RTA→RTA Control Panel。必须装入带有 DSP/BIOS 内核的 out 文件，才能打开该控制面板。在该窗口中，可以看到一系列检查框，激活或禁止各种类型的实时跟踪，见图 5.41。在 CCS 2.0 以后的版本中，如果进行默认设置，则所有类型的跟踪都处于激活状态。例如，想关闭对 SWI 线程的跟踪信息，可以清除“SWI logs”的检查框。

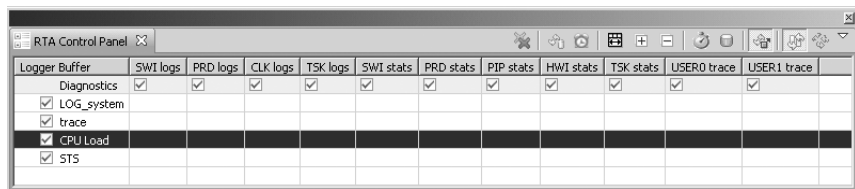


图 5.41 CCSv5 DSP/BIOS 工具控制面板

#### 2. 内核/模块查看窗口

该工具在 CCSv5 下的 DSP/BIOS 被去除了，但在 CCS3.3 以及 CCSv5 的 SYS/BOIS 下仍

然可以使用。通过内核/模块查看调试工具观察当前配置、状态和在目标板上运行的 DSP/BIOS 模块的当前状态。为了显示内核/模块查看窗口，在 CCS3.3 中单击图标选择“DSP/BIOS”→“Kernel/Object View”，见图 5.42。内核/模块查看窗口以表格的形式显示如下信息。

- KNL 显示系统内核使用信息。
- TSK 显示任务线程的运行状态。
- MBX 显示邮箱模块的使用情况。
- SEM 显示旗语模块的使用情况。
- MEM 显示存储段的使用情况。
- SWI 显示软件中断线程的运行情况。

图 5.42 所示的是一个 DSP/BIOS 程序的所有任务模块的运行状态。通过该窗口，可以了解这些任务的优先级、运行状态、堆栈使用情况等。图 5.42 所示的内核/模块查看工具窗口中，都有一个“Refresh”刷新按钮。按下每个表格右上角的该刷新按钮，CCS 将会同时刷新所有表格内的数据，即刷新的过程是同步的。当目标板上的 DSP 处于运行时，按下刷新按钮后，CCS 会自动停止目标板的 DSP 的运行，并开始收集数据，然后继续目标板 DSP 的运行。若打开了该内核/模块查看窗口，只要目标板的 DSP 停止运行，如遇到断点等，CCS 将自动进行刷新。为了方便观察，如果刷新后所显示的数据与前一次不同，则变化的数据显示为红色。

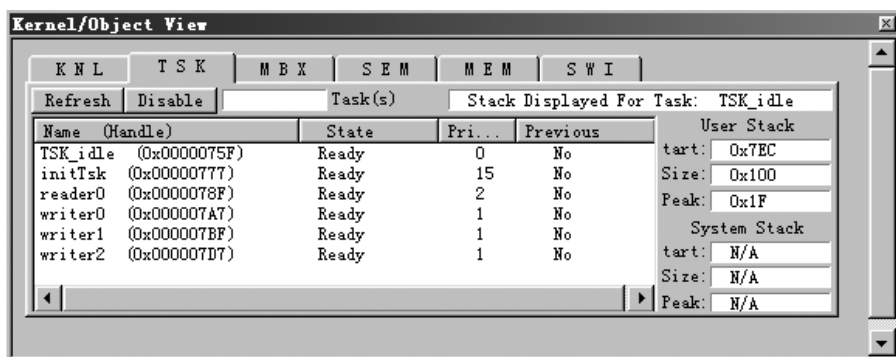


图 5.42 DSP/BIOS 内核模块查看窗口

### 3. CPU 负载图

为了显示 CPU 的负载图，可以在 CCSv5 的 Debug 窗口中选择 Tools→RTA→CPU Load，打开 CPU 负载状态图。在例 5.8 的图 5.40 中，可以看到目标板 DSP 处理负载的曲线，此时的 CPU 负载大约为 40%。CPU 负载，包括从目标板到主机，以及运行附加的后台任务所需要的所有时间。CPU 负载数据在一次查询周期内统计。但当 CPU 负载太高时，由于几乎没有时间运行后台的统计线程，会导致 CPU 负载图显示不出来。如果将例 5.8 的 volume\_bi-os.c 中的 processingLoad 变量改为 10，可能会看不到 CPU 负载图的显示。

### 4. 程序模块执行状态图

程序中各个线程执行的运行状态显示图，在 CCSV5 的 DSP/BIOS 也被去除，但 CCS3.3 下仍然保留。在 CCS3.3 打开执行状态图窗口，可以选择“DSP/BIOS”→“Execution Graph”菜单选项，见图 5.43。在这个窗口中，可以看见程序中的各个线程运行状态图。除

非在“RTA Control Panel”窗口中禁止记录各种目标类型，否则该图表会被更新。

该图包括了很多行，包括 DSP/BIOS 程序中定义的 HWI 硬件中断服务程序、SWI 软件中断、TSK 任务、旗语模块、周期函数，以及时钟模式信号。另外，还包括了内核线程和其他的 IDL 空闲线程。图 5.43 显示 initTsk 任务、KNL\_swi 内核线程以及 SEM 旗语的释放标记。另外，“Time”和“PRD Ticks”分别标记定时器中断和周期性函数执行的时刻。

图 5.43 显示的线程，优先级从高到低，使用不同颜色表示各个线程的状态。

- 白色，线程没有运行或者没有准备好运行。
- 白框，线程已经进入执行队列等待，即已经准备好运行。
- 蓝框，线程正在运行，即该线程正在使用 CPU。
- 黑框，线程运行完毕，如图中的 initTsk 任务。
- 蓝绿框，在该轮询间隔开始到结束的时间片中，没有该线程的状态信息。
- 绿框，识别程序对“LOG\_message”功能的调用，把用户信息写到系统日志中。
- 红框，有错误发生。例如，当调用 LOG\_error 功能，或执行状态图检测出某线程没有满足实时要求时，则有错误发生。无效的日志记录出现，也意味着发生了错误。无效的日志记录，可能是在程序记录系统日志时产生的。
- 红线，超过了轮询间隔。如果系统日志设置成循环状态，在主机上就看不到大量的、在轮询间隔内发生的日志消息。

图 5.43 中显示的信息，DSP/BIOS 内核将使用一段内存（DSP 的一段数据存储器）来存放。用户可以在 DSP/BIOS 的配置文件中，通过修改 LOG 模块参数，来改变存储器的长度。存储器越大，能记录的信息量也越大。由于 DSP 的数据存储器有限，所以一般将这个存储器的长度设置为 256 或 512，允许循环使用该存储器。

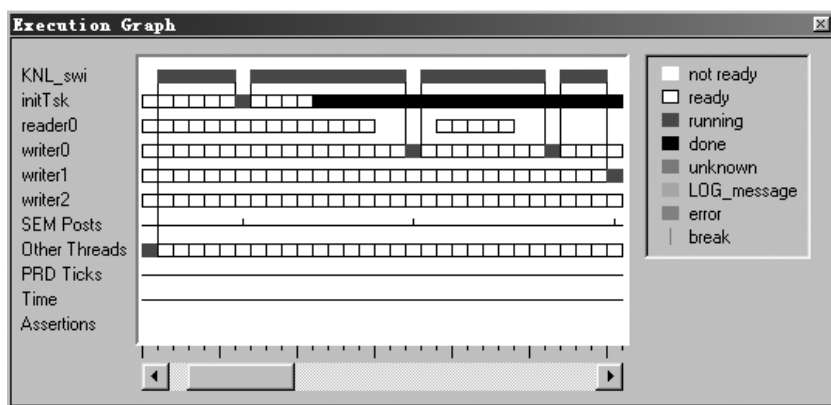
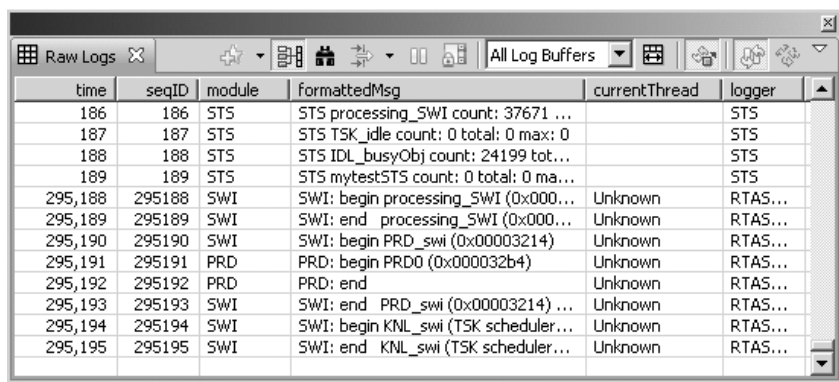


图 5.43 程序模块执行状态图

## 5. 信息显示窗口

DSP/BIOS 内核专门提供了一个信息输出窗口，用来替代标准 C 下的 stdout 窗口；使用 LOG\_printf 函数，替代标准 C 中的 printf 函数。使用 DSP/BIOS 内核提供的 LOG\_printf 函数，比标准 C 中的 printf 函数有更高的效率。CCSv5 中有两种 Log 记录窗口输出：Raw Logs 和 Printf Logs。图 5.44 是 Raw Logs 的打印信息输出显示。该窗口包含了 DSP/BIOS 代码所有的打印信息，有系统内部的，也有用户代码调用函数 LOG\_printf 打印的。而 Printf Logs 仅显示函数 LOG\_printf 打印的信息。



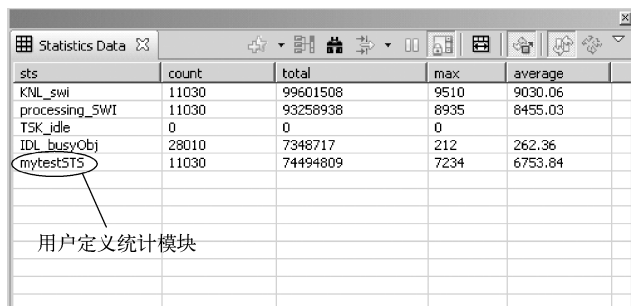


time	seqID	module	formattedMsg	currentThread	logger
186	186	STS	STS processing_SWI count: 37671 ...		STS
187	187	STS	STS TSK_idle count: 0 total: 0 max: 0		STS
188	188	STS	STS IDL_busyObj count: 24199 tot...		STS
189	189	STS	STS mytestSTS count: 0 total: 0 ma...		STS
295,188	295188	SWI	SWI: begin processing_SWI (0x000...	Unknown	RTAS...
295,189	295189	SWI	SWI: end processing_SWI (0x000...	Unknown	RTAS...
295,190	295190	SWI	SWI: begin PRD_swi (0x00003214)	Unknown	RTAS...
295,191	295191	PRD	PRD: begin PRD0 (0x000032b4)	Unknown	RTAS...
295,192	295192	PRD	PRD: end	Unknown	RTAS...
295,193	295193	SWI	SWI: end PRD_swi (0x00003214) ...	Unknown	RTAS...
295,194	295194	SWI	SWI: begin KNL_swi (TSK scheduler...	Unknown	RTAS...
295,195	295195	SWI	SWI: end KNL_swi (TSK scheduler...	Unknown	RTAS...

图 5.44 DSP/BIOS 下的信息输出窗口

## 6. 状态统计窗口

为了显示状态统计窗口，在 CCSv5 的 Debug 窗口下选择 Toos→RTA→Statistics Data 选项。为了控制数据更新的频率，可以设定查询速率。除非在“RTA Control Panel”窗口中，取消统计数据累加器类型功能，否则数据会不断更新。状态统计如图 5.45 所示。



sts	count	total	max	average
KNL_swi	11030	99601508	9510	9030.06
processing_SWI	11030	93258938	8935	8455.03
TSK_idle	0	0	0	
IDL_busyObj	28010	7348717	212	262.36
mytestSTS	11030	74494809	7234	6753.84

用户定义统计模块

图 5.45 DSP/BIOS 内核提供的模块统计窗口

图 5.45 中，统计的单位默认情况是机器时钟个数。可以看出，processing\_SWI 线程所花费的机器时钟个数平均约为 8 455 个。如果 DSP/BIOS 的配置文件和例 5.8 一致，该 SWI 线程每 1ms 调用一次，则该线程每秒钟将占用大约 8.4M 个机器时钟周期。

为了方便用户统计局部代码的运行时间，DSP/BIOS 提供了 STS 统计模块。如图 5.45 中的 mytestSTS 统计项，该项显示的就是用户自己定义的统计时间。下面通过一个例子来说明。

**例 5.9** 在代码中添加统计信息。仍以前面的例 5.8 为基础，在它的 DSP/BIOS 配置文件 bios\_5505.tcf 和 volume\_bios.c 上修改。

- 添加新统计对象：打开配置文件 bios\_5505.tcf，在 Instrumentation→STS 下添加一个新的统计模块，改名为 mytestSTS，如图 5.46 所示。
- 在 volume\_bios.c 源代码中，需要统计 load() 函数的运行时间。于是，在 processing() 函数调用 load() 函数语句的前后，各添加一个 STS 统计函数。

```
STS_set(&mytestSTS, CLK_gettime()); //新增 STS 统计函数
load(processLoad);
STS_delta(&mytestSTS, CLK_gettime()); //新增 STS 统计函数
```

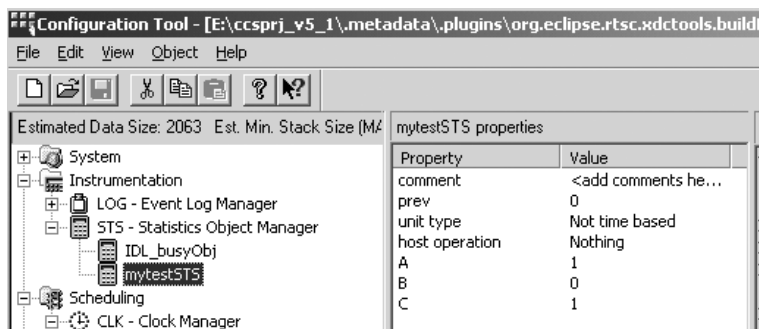


图 5.46 添加 STS 统计模块

- 在上面的代码中，使用了 BIOS 内核提供的两类管理函数：STS 统计管理与 CLK 时钟管理。STS\_set() 和 STS\_delta() 分别设置初值以及计算差值，并将计算结果放到 mytestSTS 统计模块中以便显示结果。而 CLK\_gettime() 函数是通过时钟管理模块读出系统定时器的计数值，以便统计模块计算时间差。
- 重新编译、连接、运行，打开 STS 统计窗口，便可以看到图 5.45 所示的结果。load() 每次线程运行消耗了约 6 753 个机器时钟。

### 5.3.4 XDC 工具的使用

TI 公司为了使自己及其第三方发布的软件组件（如 Codec 包等）可以重复使用，专门开发了 XDC（eXpress DSP Component）工具。XDC 工具根据一套 Build 指令，生成可执行的文件，用于实时的嵌入式系统。它包括开发 API 的工具和标准、静态配置工具和打包工具。XDC 最大的好处在于，标准化了传递过程，简化了应用程序中在目标平台下引用其他包的过程。基于 XDC 的应用开发，具有独立于硬件的标准接口、支持离线配置以便优化存储器的使用和性能的提高，并支持定制的开发环境里的自动操作的特点。

#### 5.3.4.1 XDC 工具概述

XDC 和其他编译工具（如 gmake）一样，XDC 工具可以根据编译指令生成可执行文件或者库，可以编译任何相关文件，并且可以同时为多种目标板进行编译，所编译的源文件可以为 C、C++、汇编和库文件。XDC 有其自身的特点和优势。XDC 是专门为 RTSC（Real Time Software Component）包开发的。由于 RTSC 包可以保持文件相关和版本信息，当使用 RTSC 包作为源文件时，XDC 工具可以自动进行文件相关和版本检查。

只要为应用程序提供一个简单的配置脚本，XDC 工具就可以生成代码，这在应用程序使用多个包文件时显得尤为重要，可以极大地简化诸如 Davinci Engine 和 DSP 服务器之类的复杂应用程序的编译。

下面将介绍 XDC 相关概念、以及如何安装和配置 XDC 工具。

#### ➤ XDC 的重要概念

XDC 的开发中涉及的相关概念比较多，本文主要介绍三个重要概念，其他的概念和术语，用户可以根据自己的需要参考相应的文档。

- 包（Packages）：包是对于包含模块、界面以及其他软件的通用称呼。所有的包都要

经过建立、测试、发布以及配置为一个单元。XDC 工具可以创建和使用包，针对用户来说更多的是使用 TI 以及第三方发布的 RTSC 包。这些包的内容分为两大块：负载（Payload），如库和程序；元数据（Program Metadata），如版本信息、文件依赖信息、头/库文件引用和模块（接口函数调用）等。

- 库（Repository）：库是一个装有单个或多个包的目录。库只能装有一个包的一个版本。因此，并行装入一个包的两个不同的版本，就要求分别装在两个库里。用户完全控制库的个数和名称。
- 包路径（Package Path）：包路径是定位一个包的位置时所搜索的一个库的序列。就像定位命令所使用的环境变量 PATH 一样，包路由用户设定并可以用源代码来指定包的位置，而不必使用绝对路径。简单地调整包路径，可以快速地在的一个或多个包的不同版本之间切换。

另外需要注意的是，在进行脚本文件配置时，经常需要使用一个包、界面或模块，引用中它们名字里的“.”，表示其在该库里的位置。例如，模块 ti.sysbios.knl.Task，位于 ti/sysbios/knl 路径下，也是该包库的路径。

#### ➤ XDC 的安装和卸载

开发 XDC 应用程序的首要前提是，在 PC 上安装 XDC 工具，安装的源文件可以从 TI 网站上获取。本节以 Windows 平台上 XDC\_3\_20\_08\_88 工具的安装来说明，Linux 平台上的安装可以参阅相关资料。

在 Windows 平台上，双击 xdcutils\_ccs\_setupwin32\_3\_20\_08\_88.exe 源文件，根据安装提示，选择安装路径，点击下一步，直至安装完成。至于卸载，找到 uninstall 文件，双击即可。

#### ➤ 设置环境变量

为了方便使用 XDC 工具和 xs 命令，需要将 XDC 添加到环境变量中，也就是将 XDC 的顶级安装目录添加到 Windows 平台的 PATH 定义中。根据以下步骤来设置环境变量。

- (1) 右击“我的电脑”，选择“属性”；
- (2) 在系统属性的对话框中，选择高级标签下的“环境变量”；
- (3) 选择“系统变量”中的“PATH”，点击“编辑”；在已经存在的“PATH”定义后，添加 XDC 的顶级安装目录，如；D:\EDA\xdcutils\_3\_20\_08\_88；
- (4) 点击“确定”，直至环境变量修改完成。

#### ➤ 测试 XDC 的安装

为了测试 XDC 工具的安装和配置是否成功，打开命令窗口（开始→运行→command），输入 xs --version，会看到返回：xs <XDC Script Interpreter> xdcutils - f05, Jan 25 2011 类似的语句；如果输入 xs xdc.tools.echo hello Mr. Zhao，将看到 hello Mr. Zhao 返回。这就证明 XDC 工具安装和配置成功。

### 5.3.4.2 XDC 工具调用

在使用其他编译工具时，编译工具需要知道头文件的搜索路径；连接器需要知道库文件的搜索路径。使用 RTSC 包时，如何获取这些路径信息呢？XDC 工具中 Configuro 解决了这个问题。如图 5.47 所示，生成的文件 compiler.opt 提供了编译选项和头文件路径。生成的文件 linker.cmd 提供了连接选项和库文件搜索路径。

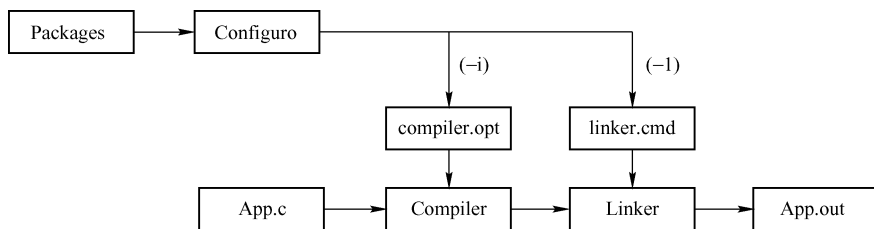


图 5.47 使用 XDC 工具的编译流程

Configuro 工具，除了需要包文件本身外，还需要其他 4 个输入。

- (1) \*.cfg: 配置文件，配置应用程序需要调用哪个软件包。
- (2) XDCPATH: 列出所有包的搜索路径。
- (3) 平台，如 ti.platforms.evmDM6437。
- (4) 目标，如 ti.targets.C64P。

使用基于 XDC 的软件包，来创建应用程序，要遵循建立在传统的 C 编程技术基础上的开发周期。XDC 包将 TI 和第三方所提供的软件标准化，更容易集成到用户的应用程序中。使用 XDC 包，要在传统的编译/链接周期里增加一个配置步骤。XDC 提供了若干种工具，以简化 XDC 包的集成和使用。使用基于 XDC 的软件包创建应用程序的基本流程如下：

- (1) 配置一个应用程序；
- (2) 编写 C 代码；
- (3) 针对用户的目标系统和平台处理该配置；
- (4) 编译和链接应用程序。

关于配置脚本文件的语句类型、C 代码的编写规范以及目标、平台的类型，用户可以查阅 TI 提供的手册。

### 5.3.4.3 基于 XDC 工具的应用程序的开发

本节将给出两个基本的例子，分别在 CCSv5.2 下用 TI 提供的 gmake.exe，来使用 XDC 工具构建应用程序。

#### 例 5.10 基于 CCSv5 实现 XDC 应用程序的构建

本应用程序的功能主要是实现对一幅 640×480 的 YUV 格式的静态图像进行 JPEG 编码。下面会详细介绍如何使用 CCSv5 构建 XDC 应用程序，读者可以根据步骤一步一步地实现。

1. 创建一个 CCSv5 的空的 RTSC 项目 (File→New→CCS Project)，项目的文件名为 Jpeg\_Encode，其他参数的配置如图 5.48 所示，然后点击 Next。

下一步，在属性选项卡中，选择 XDC 工具版本，并且配置所需要的目标和平台类型。用户也可以在此选择或者添加所需要库和包的搜索路径，当然也可以等项目创建以后再添加相应的包的搜索路径。本例子选择的是后一种方法。如图 5.49 所示，配置完成后，点击 Finish，项目文件创建就完成了。

2. 为应用程序创建一个配置脚本文件。

和 Java、C 不同，开发 XDC 应用程序需要使用 JavaScript 语法写一个配置文件，进行一些静态地配置预备工作，以便应用程序来使用 XDC 包。此外，用户的 C 程序可以进行动态的配置，但 XDC 的配置定义是起始点。配置文件具有以下作用：

- 1) 指定使用的模块和包，以及创建的静态对象；



图 5.48 创建一个 RTSC 项目 (1)

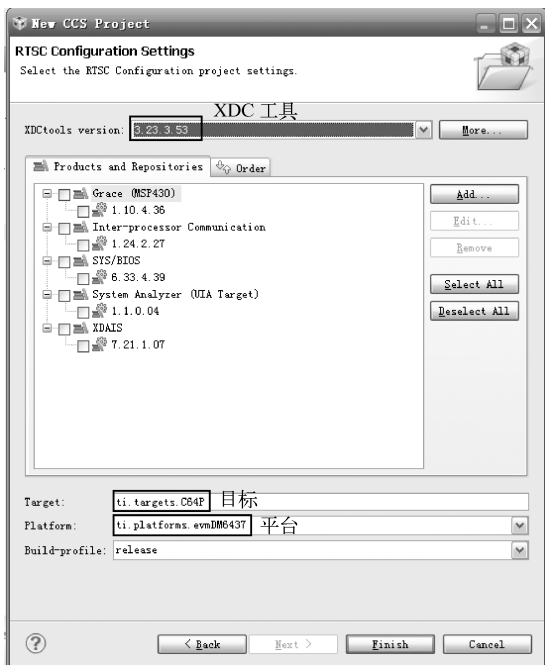


图 5.49 创建一个 RTSC 项目文件 (2)

- 2) 在指定的和依赖的包之间进行完整性检查;
- 3) 设置模块或对象的配置选项, 改变其默认属性;
- 4) 简化应用程序的编译和链接过程。

一般情况下, 该配置文件位于用户的主 C 代码的同一个目录下。它可以使用任何名字, 但名字里不能有空格, 建议使用小写字母作名字。使用一个文字编辑器来建立配置文件, 并存为一个后缀名为 .cfg 的文件, 如 mycfg.cfg。要确认存的是无格式文件 (不能是 Word 或其他文字处理器文件)。

本项目文件创建 Jpeg\_Encode.cfg 的配置文件, 主要进行了三方面的配置: Codec 配置、DSKT2 配置和 DMAN3 配置, 配置的过程如下。

首先对 OSAL 模块和 Codec 模块进行声明, 然后对创建的 codec 线程的属性进行配置, 代码如下。

```
/* set up OSAL */
var osalGlobal = xdc.useModule( ti.sdo.ce.osal.Global );
osalGlobal.runtimeEnv = osalGlobal.DSPBIOS;
osalGlobal.defaultMemSegId = "DDR2";
Program.main = Program.system = null;

/* get various codec modules; i.e., implementation of codecs */
var JPEGENC = xdc.useModule( codecs.jpeg_enc.JPEG_ENC );

/* ===== Engine Configuration ===== */
var Engine = xdc.useModule( ti.sdo.ce.Engine );
var vcr = Engine.create( "webcam", [
    { name: "jpegenc", mod: JPEGENC, groupId: 0, local: true },
]);
```

其次，对 DSKT2 进行配置，主要是把 XDAIS 使用到的 IALG memory 类型和 DSP memory 结合起来，定义缺省的 scratch 组的 memory 大小，代码如下。

```
var DSKT2 = xdc.useModule( ti.sdo.fc.dskt2.DSKT2 );
DSKT2.DARAM0      = "L1DSRAM";
DSKT2.DARAM1      = "L1DSRAM";
DSKT2.DARAM2      = "L1DSRAM";
DSKT2.SARAM0      = "L1DSRAM";
DSKT2.SARAM1      = "L1DSRAM";
DSKT2.SARAM2      = "L1DSRAM";
DSKT2.ESDATA      = "DDR2";
DSKT2.IPROG       = "L1DSRAM";
DSKT2.EPROG       = "DDR2";
DSKT2.DSKT2_HEAP  = "DDR2";
DSKT2.DARAM_SCRATCH_SIZES = [ 65536,1024,0,0,0,0,0,/* ... */ ];
DSKT2.SARAM_SCRATCH_SIZES = [ 65536,1024,0,0,0,0,0,/* ... */ ];
```

最后对 DMAN3 配置，主要是定义 DMAN3 可以管理的 DMA 通道号和 DMAN3 可以提供给算法的 TCC 号，代码如下。

```
var DMAN3 = xdc.useModule( ti.sdo.fc.dman3.DMAN3 );
DMAN3.heapInternal = "DDR2";
DMAN3.heapExternal = "DDR2";
DMAN3.paRamBaseIndex = 78;
DMAN3.numQdmaChannels = 8;
DMAN3.qdmaChannels = [ 0,1,2,3,4,5,6,7 ];
DMAN3.numPaRamEntries = 48;
DMAN3.numPaRamGroup[0] = 48;
DMAN3.numTccGroup[0] = 8;
DMAN3.tccAllocationMaskL = 0;
DMAN3.tccAllocationMaskH = 0xffffffff;
```

Jpeg\_Encode.cfg 添加到项目文件中后，在项目属性中添加 XDC 工具使用到的包的搜索路径，如图 5.50 所示，并且勾选高级选项下的“Read inline.tcf in addition to inline cfg ( --tcf)”。

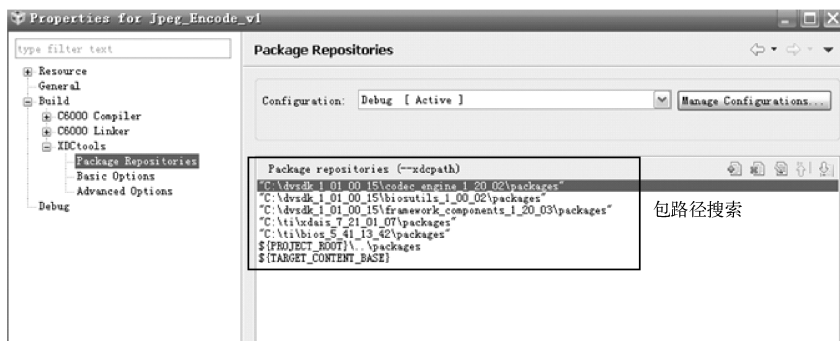


图 5.50 设置包搜索路径



配置文件完成后, 其应用程序还需要三个文件: .tcf、主程序 Jpeg\_Encode.c 和 link.cmd。

### 3. TCF 文件

创建一个 DSP/BIOS 的配置文件 Jpeg\_Encode.tcf 并添加到项目中, 同时在 TCF 文件中定义一个静态任务 JPEG\_ENCODE, 来完成图片的编码任务。此外, TCF 文件定义 DSP 的 Memory Map、设置 DSP 的复位/中断向量表, 并且创建和初始化 BIOS 程序需要的各种数据对象 (如图 5.51 所示)。

### 4. 编写应用程序源代码 Jpeg\_Encode.c 和添加 link.cmd 文件

在 Jpeg\_Encode.c 的 main 函数中调用 CERuntime\_init(), 对 Codec Engine 进行初始化后, 将 CPU 转交给 BIOS 进行调度, 然后执行 JPEG\_ENCODE 任务。在任务的子函数中调用 IMGENC\_control() 和 IMGENC\_process() 函数, 实现对图像的编码。

在 .tcf 中, 只能定义编译器默认的 sections (如 .text 和 .bss 等), 但是, 可以在 link.cmd 中定义自己的 sections, 存放程序运行中的数据。

5. 在完成以上步骤后, 对项目的头文件的搜索路径和使用到的库进行相关配置, 如图 5.52 所示。

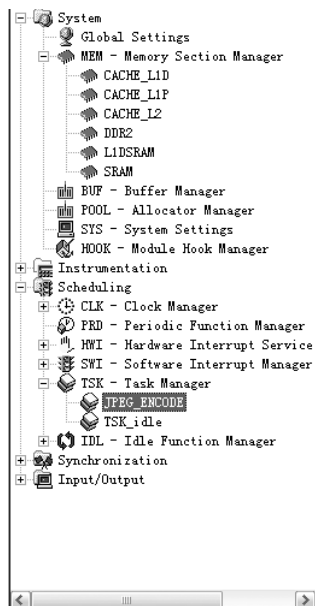


图 5.51 DSP/BIOS 配置文件

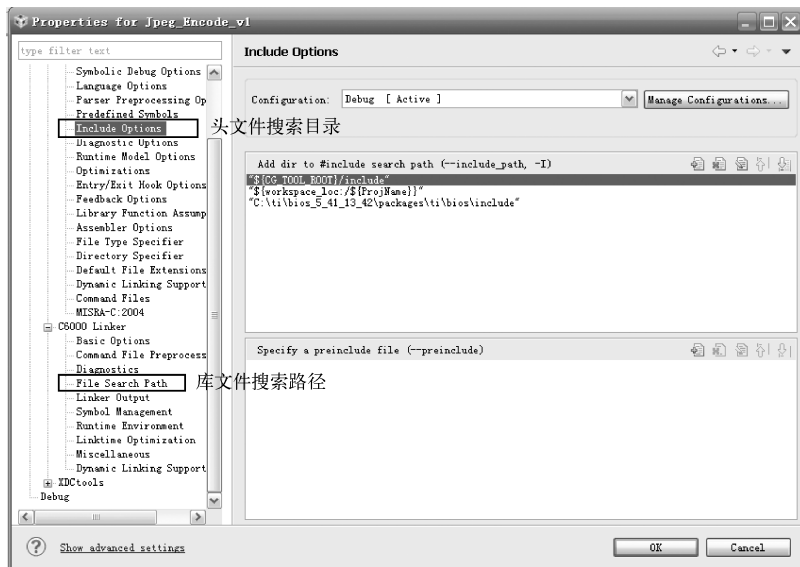


图 5.52 头文件和库文件路径配置

6. 对项目进行编译、连接, 然后将生成的 Jpeg\_Encode\_v1.out 下载到 DM6437EVM 板中, 运行, 观察编码前和编码以后的数据。编码前的数据使用 RawPlayer 查看, 编码后的数据用 Windows 自带的图片查看工具查看 JPEG 图像。

**例 5.11** 本例使用 TI 提供的 gmake 工具, 调用 Makefile 运行 configuro 工具, 构建应用

程序。该例子的功能是，在输出窗口中输出一个 Hello World!，虽然比较简单，但足以说明如何使用 XDC 工具。

首先创建一个 hello 的项目目录，然后在该目录下创建如下的文件。

### 1. 配置文件 mycfg. cfg

```
/*
 * =====mycfg. cfg =====
 */
var System = xdc. useModule( "xdc. runtime. System" );
```

### 2. 主程序代码 hello. c

```
/*
 * =====hello. c =====
 */
#include <xdc/runtime/System. h>
void main() {
    System_printf( " Hello World!" );
}
```

### 3. 编写 Makefile

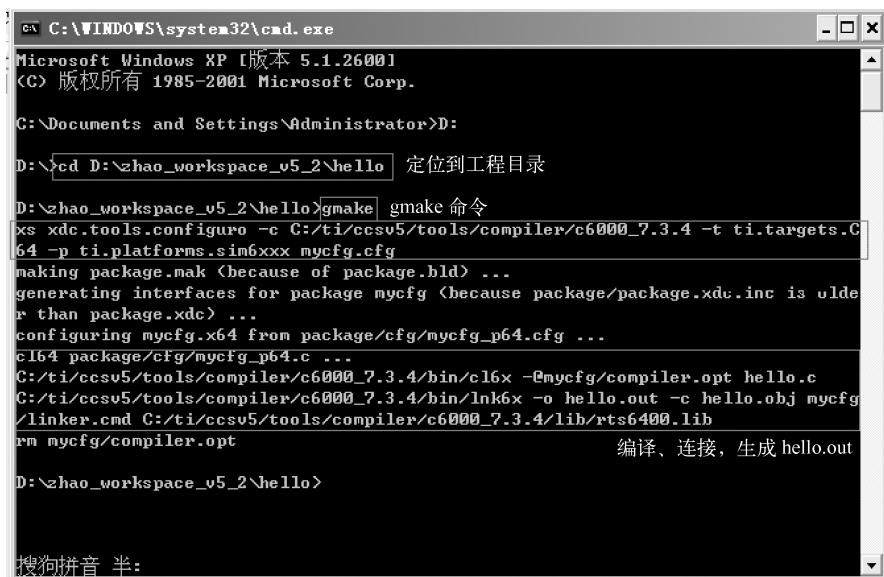
Makefile 中的内容如下。

```
CGTOOLS = C:/ti/ccsv5/tools/compiler/c6000_7.3.4
CC = $(CGTOOLS)/bin/cl6x
LNK = $(CGTOOLS)/bin/lnk6x
RTS = $(CGTOOLS)/lib/rt6400.lib
CONFIG = mycfg
XDCTARGET = ti.targets.C64
XDCPLATFORM = ti.platforms.sim6xxx
%/compiler.opt %/linker.cmd :%.cfg
    xs xdc. tools. configuro -c $(CGTOOLS) -t $(XDCTARGET) -p $(XDCPLATFORM)
$ <
%.obj:%.c $(CONFIG)/compiler.opt
    $(CC) -@ $(CONFIG)/compiler.opt $ <
hello.out:hello.obj $(CONFIG)/linker.cmd
    $(LNK) -o hello.out -c hello.obj $(CONFIG)/linker.cmd $(RTS)
```

### 4. 编译、连接

使用 CCSv5 提供的 gmake 工具进行编译和连接，在命令行下具体的编译过程如图 5.53 所示。

编译连接完成后，项目目录下的文件如图 5.54 所示。



调用XDC工具，  
生成 compiler.opt  
和 linker.cmd

图 5.53 命令行下 gmake 编译项目

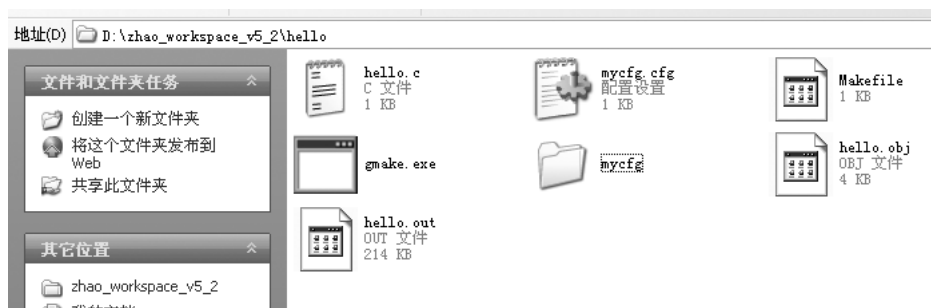


图 5.54 编译后的项目目录

## 5.4 C6EZ 工具的使用

C6EZ 工具包括 C6Run、C6Accel 和 C6Flo。C6EZ 可以帮助 ARM、Linux 的系统开发人员便捷地利用 TI 集成型浮点与定点 DSP + ARM 处理器旗下 TMS320C6000 数字信号处理器 (DSP) 的实时高密度信号处理功能。C6EZRun 与 C6EZAaccel 软件开发工具可帮助 ARM 开发人员便捷地进行 DSP 编程，不但可简化和加速开发进程，而且还可缩短 DSP 的开发启动时间与产品的上市时间，并降低开发成本。

### 5.4.1 C6Run 工具的使用

TI (Texas Instruments) 为了简化 ARM + DSP 这种双核 SoC (System On Chip) 中 DPS 程序的开发工作，C6Run (C6EZRun) 应运而生。C6Run 工具是 TI 公司推出的一款免费开源的开发工具包。程序开发者在使用双核 SoC: ARM + DSP 进行程序开发时，C6Run 工具能够帮助实现 ARM 和 DSP 处理器之间代码的无缝衔接，从而降低双核处理器应用程序的开发难

度，并减少开发时间。利用 C6Run 工具，不需要任何其他的编译工作或者软件框架来支持，就可以轻松完成 ARM + DSP 双核 SoC 平台的 C 语言程序开发。

在 C6Run 工具的底层，是将目标 DSP 执行的代码编译并重新封装成为 ARM 可执行的文件，或者是 ARM 可链接的库文件。C6Run 工具又分为两个小工具，分别是 C6RunApp 和 C6RunLib。

C6RunApp 相当于 DSP 的一种交叉编译工具。经过它重新编译的 C 代码，能够适应多种 TI 的 ARM + DSP 处理器。C6RunLib 工具所做的工作和 C6RunApp 工具类似，将在下面小节中详细介绍这两个工具。

#### 5.4.1.1 C6Run 工具的安装和配置

C6Run 工具由 TI 免费提供，下面简单介绍其安装和配置的基本步骤。

##### 1. 下载 C6Run 工具

从 TI 官方网站 <http://www.ti.com/tool/c6run-dsparmtool>，可以下载到 C6Run\_0\_98\_03\_03.tar.gz 或更新的安装包。

注：C6run 工具同时还要依赖以下软件开发包，在后面的配置中会提到并做相关配置。

- TI DSP/BIOS 或 SysBIOS Real - Time Kernel
- TI XDCtools
- TI C6000 Code Generation Tools
- TI DSP/BIOS Link
- TI Linux Utils
- TI Local Power Manager

##### 2. 安装 C6Run

在 ARM 核的 linux 的提示符下，执行命令（将 C6Run 压缩包解压到/home/user/目录下）。

```
linuxhost#cp C6Run_0_98_03_03.tar.gz /home/user
linuxhost#cd /home/user
linuxhost#tar -zxvfC6Run_0_98_03_03.tar.gz
```

注：用户可自行更改上面的/home/user 目录。

##### 3. 配置

使用编辑器打开 Rules.mak 文件，修改以下几个临时变量。

- 1) SDK\_PATH ? = \$(C6RUN\_INSTALL\_DIR)
- 2) CODEGEN\_INSTALL\_DIR ? = \$(HOME)/toolchains/TI\_CGT\_C6000\_7.3.0
- 3) ARM\_TOOLCHAIN\_PATH ? = \$(HOME)/toolchains/arm-2009q1
- 4) ARM\_TOOLCHAIN\_PREFIX ? = arm-none-linux-gnueabi-

以上指定了 C6Run 需要用到的工具的安装路径，具体路径根据工具的实际安装路径可能有所不同。在 Rules.mak 文件中，还有其他文件包需要配置，可以根据具体软件包的安装位置以及 Rules.mak 文件中的注释说明来设定。

##### 4. 编译

配置好了 Rules.mak 后，执行以下命令，完成 C6Run 工具的配置、安装工作。

```
linuxhost#cd /home/user
linuxhost#make
```

### 5.4.1.2 C6RunLib 的使用

C6RunLib 工具是 C6Run 的一部分，它的作用是将 DSP 端运行的代码封装成 ARM 应用程序可以直接调用的库函数，图 5.55 所示是一个示意图。

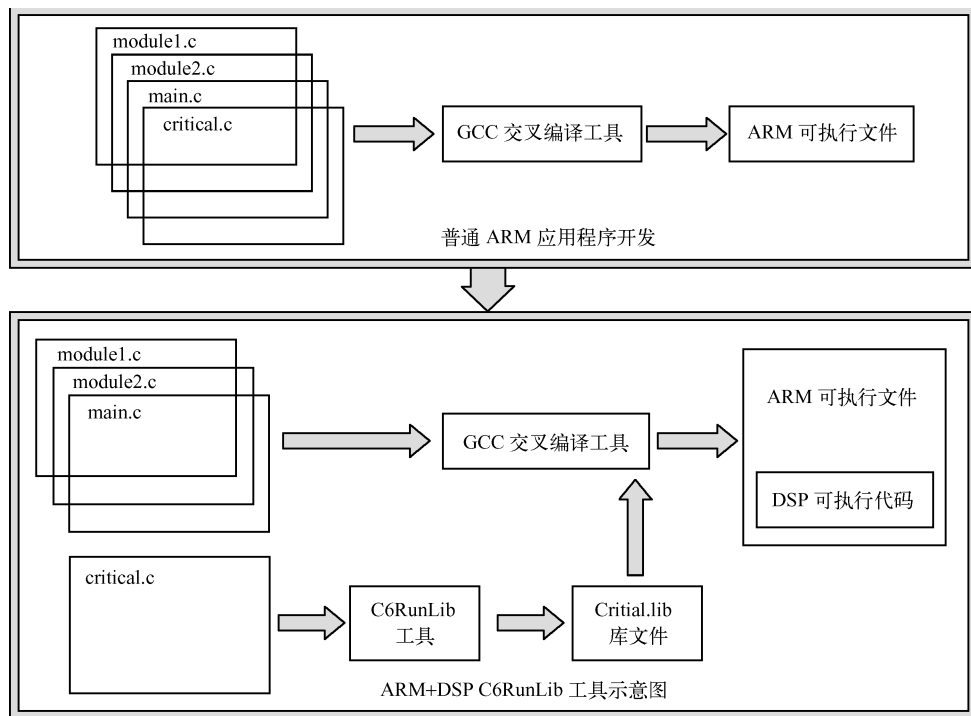


图 5.55 C6RunLib 工具的使用

下面是 C6RunLib 工具的简单用法和相应的说明。

1. 将要在 DSP 上执行的函数放在一个 c 文件里面，如 libfxns.c，并加入代码，例如：

```
int plus (int x,int y) {
    return (x + y);
}
```

显然，这是一个简单的加法运算。

2. 使用如下命令编译 libfxn.c 文件。

```
host $ c6runlib -cc -c -o libfxns.o libfxns.c
```

以上生成了 C6000 OBJ 文件 libfxns.o。

3. 使用如下命令将上面产生的 libfxns.o 文件封装成库。

```
host $ c6runlib -ar rcs libfxns.lib libfxns.o
```

将 libfxns. o 文件链接成库，并编译到 C6Run 的软件框架中，最终生成 libfxns. lib 文件。

4. 在 ARM 应用程序中调用 DSP 上运行的目标函数，例如：

```
void main( ) {
    int ret;
    ret = plus(1,2);
    printf("c = %d\n",ret);
}

host $ arm - none - linux - gnueabi - gcc - c - o main. o main. c
```

5. 使用如下命令链接上面产生的 main. o 和 libfxns. lib，完成编译。

```
host $ arm - none - linux - gnueabi - gcc - lpthread - o program. out main. o libfxns. lib
```

至此，得到了 ARM 可执行文件 program. out。将 program. out 文件复制进 linux 文件系统，运行，输出结果为 3。运算过程是在 DSP 上进行的，但是给开发者的感觉是，程序一直在 ARM 上运行。更多详细资料可访问网站：

[http://processors.wiki.ti.com/index.php/C6RunLib\\_Documentation](http://processors.wiki.ti.com/index.php/C6RunLib_Documentation)。

#### 5.4.1.3 C6RunApp 的使用

如图 5.56 所示，C6RunApp 工具是一个类似于 CG Tools（Code Generate Tools）的 DSP 交叉编译工具。C6RunApp 先将 C 语言程序编译成 C6000 OBJ 文件，并且将这些 OBJ 文件链接起来，成为一个 ARM 程序。可以看出，C6RunApp 和 CG tools 最大的区别在于，链接 OBJ 文件的时候，后者将 OBJ 文件链接成为 DSP 可执行文件，前者则链接成为 ARM 的可执行文件。

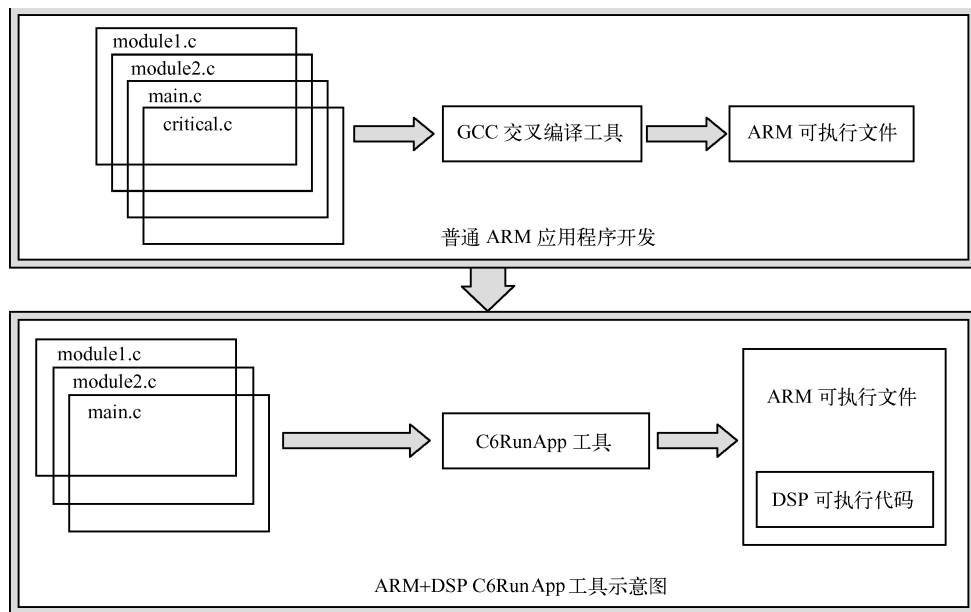


图 5.56 C6RunApp 工具的使用



下面是 C6RUNAPP 工具的简单用法。

- 将 hello\_world 文件编译成 OBJ 文件。

```
linuxhost $ c6runapp -cc -c hello_world.c
```

- 将 hello\_world.c 文件编译成默认文件名的 ARM 可执行文件。

```
linuxhost $ c6runapp -cc hello_world.c
```

- 将 hello\_world.c 文件编译成指定文件名的 ARM 可执行文件。

```
host $ c6runapp -cc -o hello_world.out hello_world.c
```

- 同时将多个文件编译成 ARM 可执行文件。

```
host $ c6runapp -cc bench.c distance.c
```

- 同时将多个文件编译成 OBJ 文件。

```
host $ c6runapp -cc -c bench.c distance.c
```

- 同时将多个文件编译成执行名字的 ARM 可执行文件。

```
host $ c6runapp -cc -o test.out bench.c distance.c
```

- 使用优化选项同时将多个文件编译成 ARM 可执行文件。

```
host $ c6runapp -cc -o3 -o test.out bench.c distance.c
```

更多详细使用方法请访问网站：[http://processors.wiki.ti.com/index.php/C6RunApp\\_Documentation](http://processors.wiki.ti.com/index.php/C6RunApp_Documentation)。

### 5.4.2 C6Accel 工具的使用

C6Accel 是由 TI 免费提供的的一个软件开发工具，C6Accel 也称 C6EZAcel，ARM 开发人员可以在 ARM + DSP 的异构双核处理器上，方便地使用 DSP 端已封装优化好的软件。

C6Accel 工具和 TI 提供的 Codec Engine 框架具有紧密的关系。Codec Engine 是 TI 发布的一种 ARM + DSP 异构双核处理器的开发框架，包含了 ARM 应用程序和 DSP 算法的开发与集成、双核间的通信机制等。从图 5.57 上可以看出，C6Accel 和 Codec Engine 二者之间的关系。C6Accel 封装在 Codec Engine 之上，相对于 Codec Engine，更方便于开发人员使用。

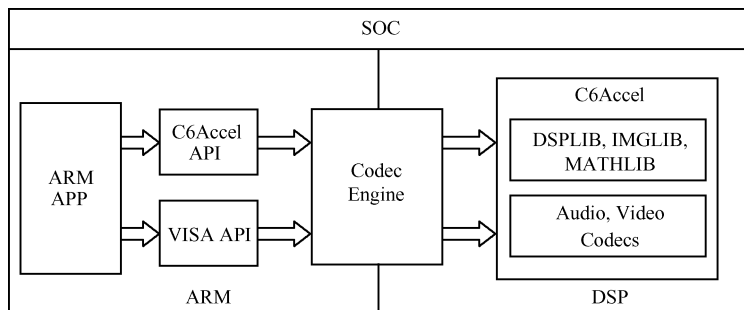


图 5.57 C6Accel 与 Codec Engine 的关系

C6Accel 包含了供 ARM 应用程序调用的 API。ARM 应用程序通过这些 API 调用 C6Accel；C6Accel API 通过 Codec Engine 接口，调用 C6Accel 集成的算法，并在 DSP 上执行。C6Accel 还简化了 ARM 应用程序的缓存管理、地址搬移、参数传递、差错处理等工作。

#### 5.4.2.1 C6Accel 的环境和配置

C6Accel 的使用目标, 是在 TI 的 ARM + DSP 异构双核处理器上开发, ARM 上运行的是 Linux。目前, C6Accel 主要支持 C6 - Integra 和 DaVinci 系列中的部分 DSP + ARM 处理器, 例如 OMAP - L138、OMAP3530、DM3730、DM816X、DM6467 等。对部分处理器的支持还处于测试之中, 用户可以在 TI 网站上查询 C6Accel 是否支持其使用的相关型号。

C6Accel 的开发环境, 取决于开发者选择的 SDK 开发环境。在基于 ARM + DSP 的双核处理器开发上, TI 提供基于 Windows 和 Linux 两种环境下的开发套件。由于目前开发者一般是在 Linux 环境下开发, 所以这里主要介绍 Linux 环境下 C6Accel 的环境配置。

##### 1. 如何获取 C6Accel

C6Accel 1. x 版本作为部分处理器 DVSDK4. x 的一部分, 集成了 codec server, 用户可以从 TI 网站上下载相关处理器的 DVSDK4. x 安装包; C6Accel 2. x 版本作为部分处理器的 EZSDK5. x 的一部分, 用户也可以从 TI 的网站上下载; 对于单独发布的 C6Accel, 用户也可以单独下载及其需要的其他资源。C6Accel 需要其他的一些相关组件, 才能构成一个最小完整系统。

对于 OMAP - L13x 和 OMAP3xxx 以及 Davinci 用户, 需要以下组件。

- Codec Engine: 2. 21 或更高版本
- XDC Tools
- XDAIS
- Linux Utils
- CODEGEN tools 6. 0. 9 或更高版本
- DSP BIOS 5. X
- BIOSUTILS
- Code sourcery tools
- Framework Components

对于 DM8148/DM8168/C6A8168 用户, 需要以下的组件。

- Codec engine 3. x
- TI CGT 7. 2
- XDCtools 3. 2x
- Linuxutils 3. 21. 00. 01
- Syslink 2. x
- Xdais 7. 2x
- SYSBios 6. 3x
- Framework components 3. 2x
- Ipc 1. 2x
- Linux - psp4. 00. 00. 07

##### 2. 安装 C6Accel

作为 SDK 的一部分, C6Accel 的配置已经包含在 SDK 的配置中, 用户在正确配置了 SDK 后, 在 SDK 根目录下编译所有组件时, 会一起将其编译。

对于用户自己下载的 C6Accel, 应该按照如下步骤进行配置和安装。

- 1) 下载 C6Accel 安装包，假设下载后放在 tmp 目录下，切换到 root，并更改权限。

```
host $ su root
host $ cd /tmp
host $ chmod +x C6Accel - x. x - Linux - x86 - Install. bin
host $ exit
```

- 2) 运行 C6Accel 安装包，进行安装。

```
host $ cd /tmp
host $ ./C6Accel - x. x - Linux - x86 - Install. bin
```

- 3) 添加所有需要组件的路径到 C6Accel 目录下的 Rules.mak 文件中，并确保要将 C6Accel 安装的路径添加在变量 C6Accel\_INSTALL\_DIR 上，例如：

```
C6ACCEL_INSTALL_DIR = <USER_DEF_PATH >
```

本步骤执行之前，要确定 Linux kernel、cmem 和 dsplink 模块已经预先生成，如果没有，用户要先编译生成这些 C6Accel 的组件。

- 4) 编译生成 C6Accel 包，进入其根目录下运行命令：

```
make all
```

- 5) 在 Rules.mak 中设置 EXEC\_DIR，生成的可执行测试应用程序将被复制到此目录下，执行下面的命令：

```
make install
```

- 6) 设备上电，打开超级终端，启动设备，进入文件系统的测试应用程序安装目录 <EXEC\_DIR> 下：

```
cd $ <EXEC_DIR >
```

- 7) 运行脚本文件 loadmodules\_c6accel\_<PLATFORM>.sh，加载 cmem 和 dsplink 模块。
- 8) 运行 test app：

```
./c6accel_app
```

#### 5.4.2.2 C6Accel 的使用

对于只关注于 ARM 应用程序的开发人员，在用 C6EZAaccel 开发时，一般只使用现成的 API 调用 C6EZAaccel 封装好的内核。而对于关注 C6Accel 整个框架的开发人员来说，还需要做 DSP 侧的算法开发与集成、ARM 的 API 开发等工作。下面分别进行简单介绍。

##### 1. C6Accel API 的使用

C6Accel 将内核划分为 7 个基本的类型：Digital Signal processing、Image Processing、

Math、Analytics、Medical、Audio/Speech processing 以及 Power/Control。对于各自的 C6Accel API 函数，用户需要详细学习。

在 C6Accel1.01.00.06 版本的/soc/app 目录下，有一个自带的测试程序例子，其中使用了 API 函数 C6accel\_IMG\_histogram\_8()，其功能是调用计算图像颜色直方图的内核。下面以此为例，介绍 API 函数在 ARM 的应用程序中如何使用。

- 1) 需要包含 Codec Engine 和 C6Accel 的头文件，在 appMain.c 中包含了：

```
#include <ti/sdo/ce/Engine.h>
#include <ti/sdo/ce/CERuntime.h>
#include "../c6accelw/c6accelw.h"
```

- 2) 需要声明 C6accel 句柄：

```
C6accel_Handle hC6 = NULL;
```

- 3) 定义一个 Engine 名（要和.cfg 文件中配置的不同）、算法名以及项目名。

```
#define ENGINENAME "omap138"
#define ALGNAME "c6accel"
#define APPNAME "app"
.....
static String algName      = ALGNAME;
static String engineName   = ENGINENAME;
static String progName     = APPNAME;
```

和 c6accel\_app.cfg 文件中的定义的 Engine 名字一样。

例如，在/omap138 下的 c6accel\_app.cfg 文件中：

```
var demoEngine = Engine.createFromServer(
    "omap138",
    "../c6accel_omap138.x674",
    "ti.c6accel_unitserver.omap138"
);
```

这里的“omap138”就是 Engine 名。用到的 server 是/c6accel\_omap138.x674，路径是“ti.c6accel\_unitserver.omap138”。此 server 中定义的 c6accel 变量，集成了 c6accel 求直方图的算法。

- 4) 在 C6Accel API 使用之前，要用函数 CE\_Runtime\_init() 来初始化 Codec Engine。

- 5) Codec Engine 初始化后，用户可以调用 C6accel\_create()，产生一个 C6accel 句柄。

```
hC6 = C6accel_create( engineName, NULL, algName, NULL );
```

这里就为 algName 创建了算法实体。

- 6) 一旦 C6accel\_create() 被成功调用，用户就可以用 C6Accel API 调用内核。C6accel 接口函数提供了一组 c6accel\_test\_ \* \* ( ) 的函数，这些函数中调用了 C6Accel 中的算法。

例如, 在 `c6accel_test_IMG_histogram(hC6, WIDTH, HEIGHT)` 中, 调用函数 `C6accel_IMG_histogram_8(hC6accel, pSrcBuf_16bpp, inputWidth * inputHeight, (unsigned short *)pWorkingBuf_16bpp, (unsigned short *)pOutBuf_16bpp)`, 完成直方图统计功能。

7) C6Accel 函数实体不再需要 Codec, 可以使用 `C6accel_delete()` API 删除。

## 2. 添加新的库/核到 C6Accel 中

用户可以添加自己常用的核/库, 其集成流程如图 5.58 所示。内核含义相当于一个独立完整模块, 如用户自己写的 DSP 函数或者 DSP 算法, 可以由 ARM 侧应用程序通过 C6Accel 的 API 调用, 在 DSP 上执行。所有的核都被分类归纳到不同的库, 所有的库编译后, 集成为 C6Accel 算法库, 最后再生成一个 Codec, 由 ARM 应用程序调用。

下面通过 TI 提供的一个将复数拆分成实部和虚部的简单内核, 来演示如何添加新核到 C6Accel 中。这个核只有一个函数: `complxtoarealnmig()`, 从属于 DSPLIB 库。在 `c6accel_1_01_00_06` 下, 按照如下步骤进行。

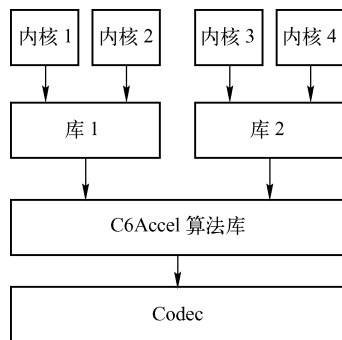


图 5.58 C6Accel 内核集成流程

### 1) 添加源程序

➤ 在 `$(C6ACCEL_INSTALL_DIR)/dsp/alg/src` 目录下添加算法 C 文件。

`complxtoarealnmig.c` 文件包含算法函数 `complxtoarealnmig()`。

➤ 在 `$(C6ACCEL_INSTALL_DIR)/dsp/alg/include` 下的 `C6accel.h` 头文件中添加函数声明。

➤ 添加用户需要的库; 在 `$(C6ACCEL_INSTALL_DIR)/dsp/alg/pjt` 目录下 `makefile` 文件中添加: `LD_LIBS += -l $(LIBRARY_PATH)/USER_LIBRARY.lib`, 或直接在 “`LD_LIBS =`” 语句后面追加亦可。

2) 确定唯一的函数 ID (function ID)。`iC6Accel_ti.h` 中定义了函数 ID 的格式, 用户必须与其保持一致。函数 ID 在 `iC6Accel_ti.h` 中创建, 由于内核确认其使用函数 ID 的低 16-bit, 因此还要在 `C6Accel.h` 文件中创建一个等效的函数 ID, 其值是函数 ID 的高 16-bit 和 0 相与。

注意到 `iC6Accel_ti.h` 是算法和应用程序的接口头文件, 由算法和应用程序共享; 而 `C6Accel.h` 只在算法中使用。

函数 ID `0x01010410`。其构成是: 31~24-bit 为提供商 ID (Vendor ID); 23~16-bit 为库 ID, 15~0-bit 为函数在其库内的 ID。由于 `complxtoarealnmig()` 函数的提供方 TI (0x01), 从属于 DSPLIB (0x01), 在库内的 ID 是 `0x0410`。所以其函数 ID 在 `iC6Accel_ti.h` 中按如下定义:

```
#define DSPF_COMPLXTOREALNIMG_FXN_ID 0x01010410
```

在 `C6Accel.h` 中, 等效的函数 ID 为:

```
#define F_COMPLXTOREALNIMG_FXN_ID 0x00000410
```

用户也可创建自己的库 ID, 如这里的 DSPLIB 是 0x01, 函数在库内的 16-bit ID 一般由用户自己定义, 但不能和其他函数的 ID 相同。为此, 可以在 `iC6Accel_ti.h` 中看到各函数 ID

的列表。

3) 创建输入参数的结构体。XDAIS 算法中所有函数都有各自的输入参数结构体, 其各自域的多少, 取决于在调用 DSP 核所需要的变量个数。输入输出的数组, 都通过接口的 inBuf 和 outBuf 传递; 输入参数结构体只需要传递输入/输出数组的描述 ID。

函数调用为:

```
complxtoarealning ( float * complex_src, float * real_dst, float * img_dst, int n_elements )
```

则在 iC6Accel\_ti.h 中输入参数结构体为:

```
typedef struct DSPF_complxtoarealning_Params {
    unsigned int  cplx_InArrID1;
    unsigned int  real_OutArrID1;
    unsigned int  img_OutArrID2;
    int  n;
}
DSPF_complxtoarealning_Params;
```

4) 根据函数 ID 来调用 DSPLIB 中的 complxtoarealning() 函数。在 \$(C6ACCEL\_INSTALL\_DIR)/dsp/alg/C6ACCEL\_TI\_dsplibFunctionCall.c 内的 C6ACCEL\_TI\_dsplibFunctionCall() 函数, 会根据 DSPLIB 库中函数 ID 调用其库内各函数, 用户在此函数中添加关于 complxtoarealning() 函数的 case 语句。

```
case ( F_COMPLXTOREALNIMG_FXN_ID ):
{
    DSPF_complxtoarealning_Params * C6ACCEL_TI_DSPF_complxtoarealning_paramPtr;
    C6ACCEL_TI_DSPF_complxtoarealning_paramPtr = pFnArray;
    if ( ( ( C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> cplx_InArrID1 ) > INBUF15 ) |
        ( ( C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> real_OutArrID1 ) > OUTBUF15 ) |
        ( ( C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> img_OutArrID2 ) > OUTBUF15 ) |
        ( ( C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> n ) < 0 ) )
    {
        return( IUNIVERSAL_EPARAMFAIL );
    }
    else
    complxtoarealning ( ( float * )
        inBufs -> descs[ C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> cplx_InArrID1 ]. buf,
        ( float * )
        outBufs -> descs[ C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> real_OutArrID1 ]. buf,
        ( float * )
        outBufs -> descs[ C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> img_OutArrID2 ]. buf,
        C6ACCEL_TI_DSPF_complxtoarealning_paramPtr -> n
    );
}

break;
```



5) 生成 C6Accel 库。上面已经将内核配置完成, 在/dsp/alg/pjt 下编译和生成的 C6Accel 库放在/dsp/alg/lib 下, 将其复制到/soc/packages/ti/c6accel/lib 下, 并在此目录下用 make 命令生成 Codec。

6) 可以用 QualiTI tool 检测生成的 Codec 是否满足 XDAIS 标准, 并将 Codec 集成到 Server 中去。

7) 为内核在 ARM 侧建立 API 函数, 使得 ARM 应用程序可以调用此 API 函数, 在 \$(C6ACCEL\_INSTALL\_DIR)/soc/c6accelw/c6accelw.c 中:

```
Int C6accel_DSPF_complexcorealnimg( C6accel_Handle hC6accel, float * ptr_x,
    float * ptr_y, float * ptr_r, int npoints )
{
    XDM1_BufDesc          inBufDesc;
    XDM1_BufDesc          outBufDesc;
    XDAS_Int32            InArg_Buf_size;
    IC6Accel_InArgs       * CInArgs;
    UNIVERSAL_OutArgs     uniOutArgs;
    Int status;
    .....

    /* Set function Id and parameter pointers for first function call */
    CInArgs->fxn[0]. FxnID    = DSPF_COMPLXTOREALNIMG_FXN_ID;
    .....

    if ( hC6accel->callType == ASYNC )
    {
        .....

        else {
            .....

            status = UNIVERSAL_processAsync( hC6accel->hUni, &inBufDesc, &outBufDesc,
                NULL, ( UNIVERSAL_InArgs * ) CInArgs, &uniOutArgs );
            }
        }
    }
    else {
        .....

        status = UNIVERSAL_process( hC6accel->hUni, &inBufDesc, &outBufDesc, NULL, ( U-
            NIVERSAL_InArgs * ) CInArgs, &uniOutArgs );
        .....
    }
    .....
}
```

上面简要写出了此 API 函数, 可以发现, 其调用内核时的两个关键点。

一是用到了内核的函数 ID。

```
CInArgs->fxn[0]. FxnID    = DSPF_COMPLXTOREALNIMG_FXN_ID;
```

二是其核心还是用 `UNIVERSAL_processAsync()` 或 `UNIVERSAL_process()` 这两个 Codec Engine 中的 API 函数。

如果用户不为自己的内核创建应用程序 API，可以直接用 `UNIVERSAL` 类 API 进行调用。具体可以参考 C6accel 的相关文献，或者访问：<http://www.ti.com/tool/c6accel-dslibs>。

### 5.4.3 C6Flo 工具的使用

德州仪器 (TI) C6EZFlo 是一款免费的图形软件开发工具，此款工具可与 TI 的 Code Composer Studio IDE™ 或基于 DSP 的开发工具配合使用。C6EZFlo 能让缺乏 DSP 特定编程知识的开发者，也能创建出用于 TI DSP 的纯 C 代码应用程序。这些应用程序可按原样使用或作为原型来加快传统 C 语言的开发速度。

C6EZFlo 提供了一个直观的拖放界面，可用于创建系统方框图。图中的块可以表示从优化的 DSP 算法到外设 I/O 驱动程序等各种内容。该工具可对系统方框图进行解析，以生成结构清晰、评论良好的 C 代码；开发人员可将该代码按原样使用，或在 Code Composer Studio IDE 中进行修改。开发人员可以通过 C674x 等 TI TMS320C6000™ DSP 器件以及基于 C6000™ 平台的 DaVinci™ 数字媒体处理器（包括 DM643x 和 DM648）来充分利用 C6EZFlo，从而简单快捷地实现对所有终端应用的原型设计。

标准 C6Flo 安装中，包含可用于 DSP 系统的数十个信号处理模块。该工具还可以扩展：标准模块可以修改或用作模板，以创建自定义模块。模块提供了可用于生成应用程序 C 源码的源模板文件。

#### 5.4.3.1 C6Flo 工具的安装和配置

在使用 C6Flo 之前，必须先在 PC 上安装如下软件。

- C6Flo 图形开发工具
- CCS 集成开发工具（版本 3.3 或更高版本）
- DSP/BIOS 系统工具（版本 5.41 或更高版本）
- C6000 编译工具（版本 6.1.9 或更高版本）
- XDC 工具（版本 3.20 或更高版本）
- Microsoft .NET Framework（版本 2.0 或更高版本）

C6Flo 工具安装使用相关资料可以参见：

<http://processors.wiki.ti.com/index.php/C6Flo>。

CCSv5 自带 C6Flo 工具。

#### 5.4.3.2 C6Flo 工具的使用

在 CCS 中使用 C6Flo 工具，需要与普通的 CCS 项目文件配合使用。下面介绍两种方法，用来创建包含 C6Flo 工具的 CCS 项目文件。

##### 1. 创建空图表

(1) 点击 `File`→`New`→`CCS Project`，建立一个新的 CCS 项目。依次通过创建向导创建好项目以后，新建项目文件就会出现在 `C/C++ Projects` 桌面。注意，目前 C6FLO 工具只支持

C6000 系列。

(2) 选择新建项目文件, 并使用 File→New→Other, 或 Ctrl + N, 添加新文件。依次通过新建文件安装向导, 选择“C6Flo Diagram”作为文件的类型并指明文件名, 见图 5.59。

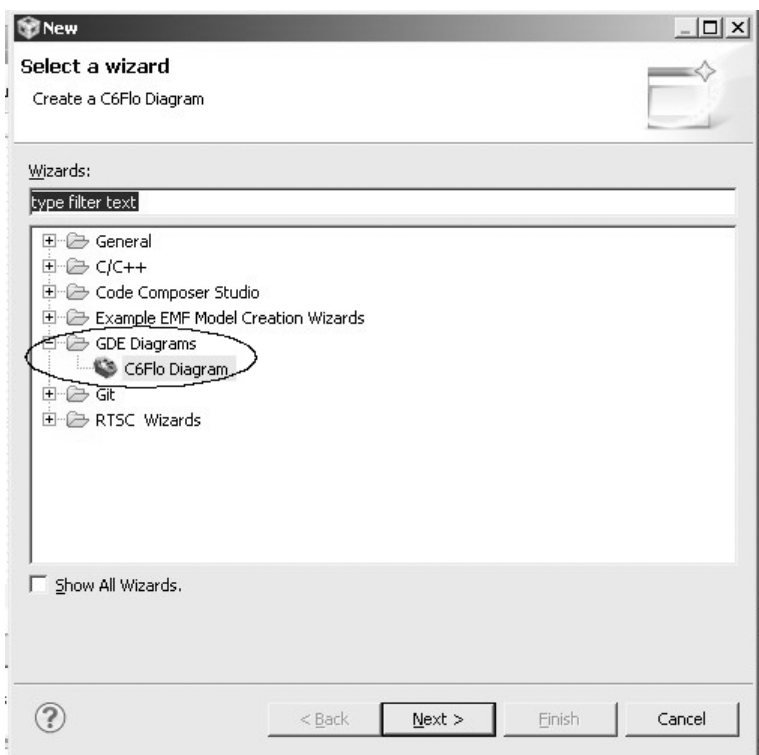


图 5.59 创建 C6Flo 空图表

## 2. 创建模板图表

(1) 点击 File→New→CCS Project, 建立一个新的 CCS 项目。依次通过项目创建向导, 直到“Project Settings”。选择应用程序最终连接的 DSP 设备。点击 Next 进入到模板图表项目, 可以看到一系列的与 DSP 设备匹配的 C6Flo 模板图表, 选择“DM6437 Audio Passthru System”。

(2) 选择与应用程序相匹配的参数配置, 点击 Finish, 一个新的项目文件将会添加到工作空间, 其后缀名为 \*.c6flo。在 C6Flo 图表编辑器中双击这个文件, 就可以修改并扩展图表, 来创建需要的 DSP 应用程序, 见图 5.60。

下面介绍使用 C6Flo 工具, 在 DM6437EVM 板上创建音频直通系统。

(1) 如果已经在 C6Flo 中创建了模板图表, 就可以直接修改各个功能块来扩展已有的 DSP 系统; 如果创建的是空图表, 需要首先设置框架块, 告诉 C6Flo 将要连接的 DSP 设备的一些基本信息, 并将此框架块拖拽到图表中。

(2) 示例采用了一种音频直通系统。在 C6Flo 工具的右边, 有相应的信号发生模块、数学计算模块, 以及滤波器模块等, 可以添加修改示例系统, 见图 5.61。

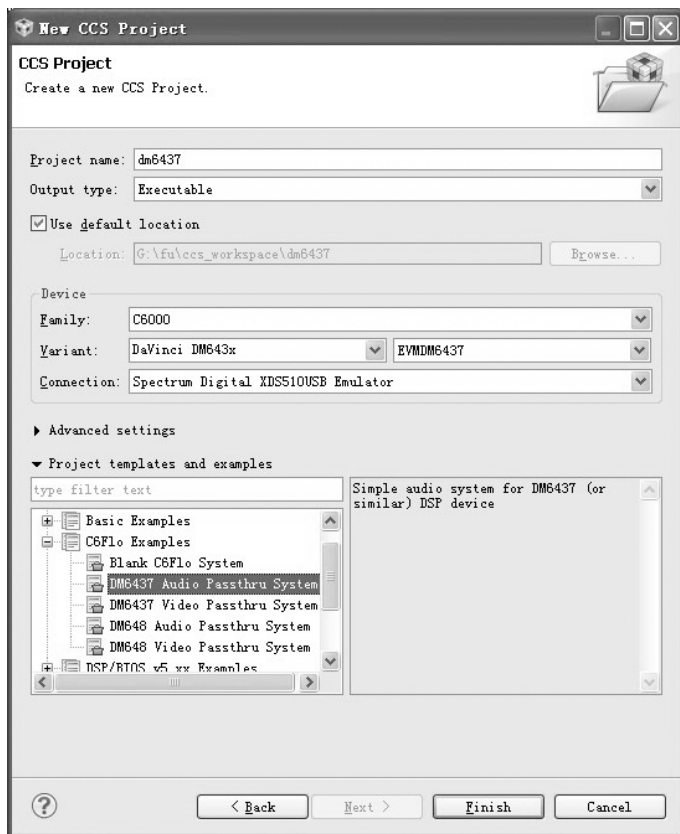


图 5.60 创建 C6FLo 模板图表

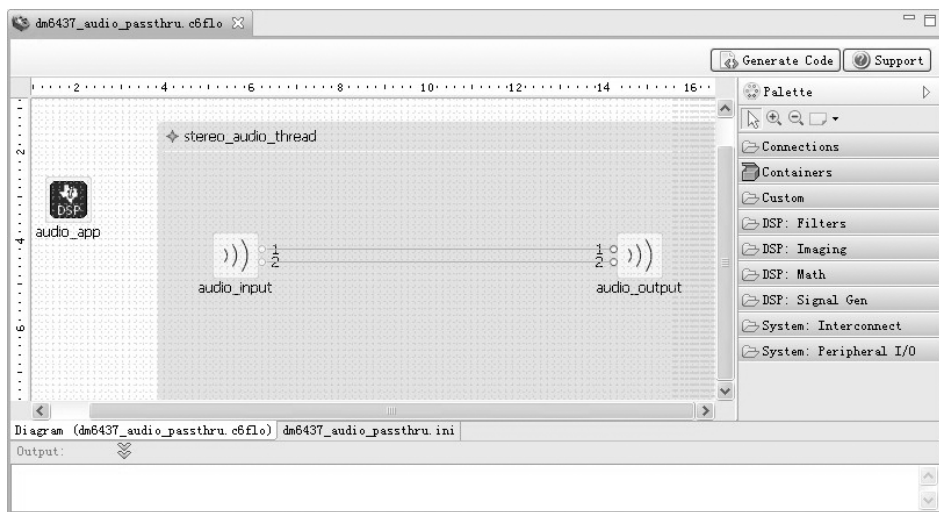


图 5.61 DM6437 音频直通系统

(3) C6FLo 允许配置各个模块参数。例如，可以配置 FIR 滤波器的处理区间和滤波器阈值。配置模块参数首先需要选中模块，点击鼠标右键，选择“Show Property View”，弹出属性窗口，修改模块参数，见图 5.62。

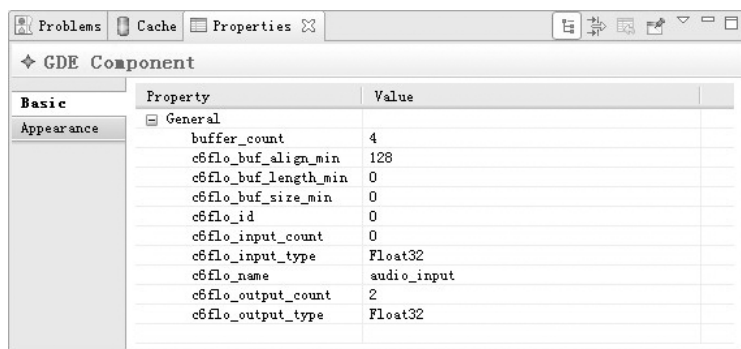


图 5.62 DM6437 音频系统参数配置

(4) 完成系统构建后, 就可以创建应用程序代码。点击“Generate Code”按钮, C6Flo 就会生成相应的 C 代码应用程序。调试编译应用程序, 下载到连接的 DSP 板上, 即可执行有关的处理。

关于 C6Flo 工具使用的详细资料可以打开 CCSv5.1, 点击“Help”, 选择“Help Contents”, 再展开“C6Flo”列表, 即会有详细介绍。或者访问网站: <http://www.ti.com/tool/c6flo-dsptool>。

#### 5.4.4.3 C6Flo 的应用例子

下面介绍一个在 C6Flo 图形开发环境中开发的典型的音视频处理系统。

(1) 点击 File→New→CCS Project, 配置相应参数, 建立一个新的 CCS 项目。

(2) 在新建项目文件下使用 File→New→Other, 或 Ctrl + N, 添加新文件, 选择“C6Flo Diagram”作为文件的类型并指明文件名。

(3) 在右边的“System: Framework”中, 选择相应框架, 拖拽到框图中, 弹出相应调色板, 包含了“App: Audio”、“DSP: Filters”、“DSP: Math”等模块, 实现了音视频发生接收, 以及中间信号的处理等功能。

(4) 完成各模块的参数配置, 并将各模块按信号的处理流程依次连接起来, 点击“Generate Code”, 生成相应的应用程序 C 代码。示例框图如图 5.63 所示。

(5) 代码生成后, 再调试编译应用程序, 生成“.out”文件。连接好相应的 DSP 板, 将编译好的应用程序下载到板上运行, 即可实现希望的处理功能。如图 5.64 所示是生成的应用程序框架。

以上是应用程序的框架结构, 它是一个简单的拥有三个源文件的 C 语言应用程序, 一个头文件、一个 DSP/BIOS 配置文件 (TCF) 和一个 CCS5.1 项目文件, 每个文件的名称和内容如表 5.3 所示。

表 5.3 文件名称和内容

文件名称	文件内容
<app>_main.c	应用程序主函数和一些通用代码, 主函数为应用程序的每一个线程动态地创建一个 DSP/BIOS TSK (包括主机控制的线程)
<app>_threads.c	用于处理和主机控制线程的回环函数, 每个回环函数包括一个 DSP/BIOS TSK, 在主函数中可动态创建的 DSP/BIOS TSK, 这些函数调用块函数来实现在框图中画出的处理链条

续表

文件名称	文件内容
<app>_blocks.c	块实例结构体和函数，每个块由创建的函数表示：创建、初始化、运行和控制，在同一个块类型的不同实例之间这些函数可能被重复使用
<app>.h	一般包含宏定义和宏命令，以及块的定义和函数原型
<app>.tcf	DSP/BIOS 配置文本，包含了框架块的大部分参数配置，在 CCS 中使用 built-in 工具，或者纯文本可以直观地观看这个文件
<app>.pj1	CCS3.3 项目文件

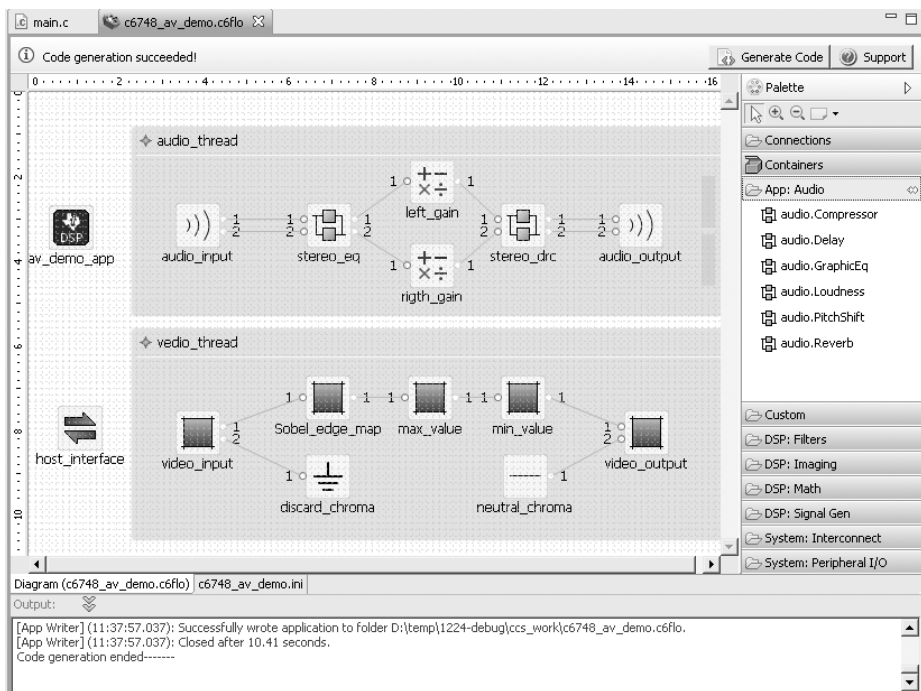


图 5.63 音视频处理框架

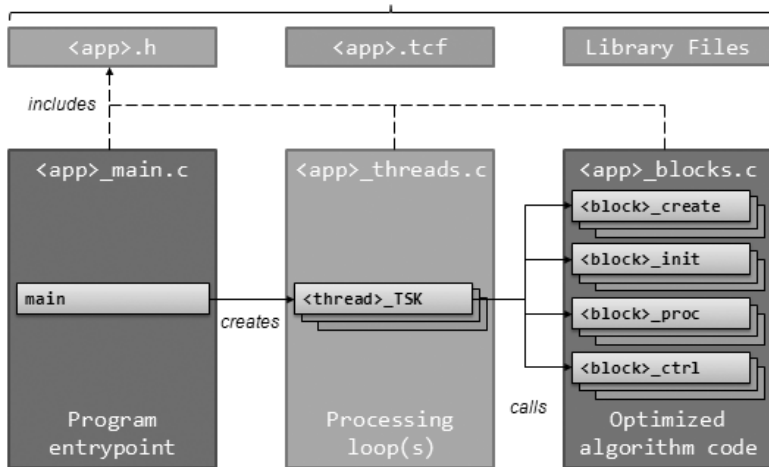


图 5.64 C6480 工具创建的应用程序框架



在音频处理系统中, 实现了采样率为 44.1kHz 的音频信号的输入, 经过均衡器处理, 再通过快速 Fourier 变换, 最后通过音质控制模块处理, 输出到音频输出模块。此系统通过对音频信号的频率特性的修正和补偿, 获得了较好的立体声处理效果。

在视频处理系统中, 可以实现标清视频信号的输入。图像信号首先经过索贝尔边缘检测模块处理, 再经过门限最大阈值和最小阈值处理模块, 输出到视频输出模块。此系统实现了视频信号中图像信号和音频信号的分路处理, 获得了较好的音、视频处理框架以及应用 DEMO。

## 参 考 文 献

- [1] Texas Instruments. ARM Assembly Language Tools v5.1 User's Guide (Rev. K) . Texas Instruments Incorporated, August 2012.
- [2] Texas Instruments. ARM Optimizing C/C++ Compiler v5.2 User's Guide. Texas Instruments Incorporated, November 2014.
- [3] Texas Instruments. ARM Optimizing C/C++ Compiler v5.2 User's Guide. Texas Instruments Incorporated, November 2014.
- [4] Texas Instruments. TMS320C54x Optimizing C Compiler User's Guide. Texas Instruments Incorporated, October 2002.
- [5] Texas Instruments. TMS320C6000 Optimizing Compiler v7.6 User's Guide. Texas Instruments Incorporated, March 2014.
- [6] Texas Instruments. TMS320C55x Chip Support Library API Reference Guide (Rev. J) . Texas Instruments Incorporated, September 2004.
- [7] Rishi Bhattacharya. Programming the TMS320VC5503/C5506/C5507/C5509/C5509A I2C Peripheral (Rev. A) . Texas Instruments Incorporated, September 2008.
- [8] Texas Instruments. TMS320C54x Chip Support Library API Reference Guide (Rev. D) . Texas Instruments Incorporated, May 2003.
- [9] Texas Instruments. DSP/BIOS 5.40 Textual Configuration (Tconf) User's Guide. Texas Instruments Incorporated, February 2009.
- [10] Texas Instruments. TMS320C6000 DSP/BIOS User's Guide. Texas Instruments Incorporated, March 2004.
- [11] Texas Instruments. TMS320C54x DSP/BIOS User's Guide. Texas Instruments Incorporated, May 2000.
- [12] Texas Instruments. TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide. Texas Instruments Incorporated, August 2012.
- [13] Texas Instruments. TMS320C55x DSP/BIOS 5.x Application Programming Interface (API) Reference Guide. Texas Instruments Incorporated, August 2012.
- [14] Texas Instruments. TMS320C6000 DSP/BIOS 5.x Application Programming Interface (API) Reference Guide. Texas Instruments Incorporated, August 2012.
- [15] Texas Instruments. TMS320C28x DSP/BIOS 5.x Application Programming Interface (API) Reference Guide. Texas Instruments Incorporated, August 2012.

- [16] Texas Instruments. XDC Consumer User's Guide. Texas Instruments Incorporated, July 2007.
- [17] Zhengting He. Creating a TMS320DM6446 Audio Encode Example Using XDC Tools. Texas Instruments Incorporated, February 2008.
- [18] Texas Instruments. Codec Engine Algorithm Creator User's Guide. Texas Instruments Incorporated, September 2007.
- [19] Joe Coombs. Using the C6Flo Graphical Development Tool. Texas Instruments Incorporated, August 2010.
- [20] Rahul Prabhu. Using C6Accel: ARM Access to DSP Software on TI SoCs. Texas Instruments Incorporated, September 2010.
- [21] Daniel Allred. C6Run Software Development Tool. Texas Instruments Incorporated, September 2010.